PROGRAMMING MODELS AND SCHEDULING TECHNIQUES FOR HETEROGENEOUS ARCHITECTURES

Judit Planas Carbonell

PhD Thesis

Departament d'Arquitectura de Computadors - DAC
Universitat Politècnica de Catalunya - UPC

Advisors:
Eduard Ayguadé
Rosa M. Badia

July 2015

Abstract

There is a clear trend nowadays to use heterogeneous high-performance computers, as they offer considerably greater computing power than regular homogeneous CPU systems. Extending regular CPU systems with specialized processing units (accelerators such as GPGPUs or Intel Xeon Phi) has become a tremendous revolution in the High Performance Computing world. Not only the traditional *performance per Watt* ratio has been increased with the use of such systems, but also the *performance per Euro/Dollar* has significantly been raised.

Heterogeneous machines can adapt better to different application requirements, as each architecture type offers different characteristics. Thus, in order to maximize application performance in these platforms, applications should be divided into several portions according to their execution requirements. These portions should then be scheduled to the processing unit that best fits their requirements.

Hence, heterogeneity introduces complexity in application development, up to the point of forming a programming wall: on the one hand, source codes must be adapted to fit new architectures, and sometimes they must even be rewritten from scratch. On the other hand, resource management becomes more complicated. For example, multiple memory spaces may exist and require explicit memory movements and, moreover, additional synchronization mechanisms must be added between different code portions that run on different processing units. For all these reasons, efficient programming and code maintenance in heterogeneous systems has been extremely complex and expensive.

Although several approaches have been proposed for accelerator programming, like CUDA and OpenCL, these models do not solve the aforementioned programming challenges, as they expose low level hardware characteristics to the programmer. Ideally, programming models should be able to hide all these complex accelerator programming by providing a homogeneous development environment.

In this heterogeneous context, this thesis makes two major contributions: first, it proposes a general design to efficiently manage the execution of heterogeneous applications and second, it presents several scheduling mechanisms to spread application execution among all the processing units of the system and maximize performance and resource utilization.

Regarding the first contribution, this work proposes an asynchronous design to manage execution, data movements and synchronizations on accelerators. This approach has been developed in two steps: first, a semi-asynchronous proposal and then, a fully-asynchronous proposal in order to fit contemporary hardware restrictions. The experimental results from different multi-accelerator platforms showed that these approaches could reach the maximum expected performance. Even if compared to native, hand-tuned codes, they could get the same results and outperform native versions in selected cases.

Regarding the second contribution, four different scheduling strategies are presented. They combine different aspects of heterogeneous programming to minimize the execution time of applications. For example, minimizing the amount of data shared between processing units and their local memory spaces, or maximizing resource utilization by scheduling each portion of code on the processing unit that fits better. The experimental results were performed on different heterogeneous platforms, including general purpose CPUs, GPGPU and Intel Xeon Phi accelerators. As shown in these tests, it is particularly interesting to analyze how all these scheduling strategies can impact application performance.

Three general conclusions can be extracted from the research work derived from this thesis. First, application performance is not guaranteed across new hardware generations. Therefore, source codes must be periodically updated as hardware characteristics evolve. Second, the most efficient way to run an application on a heterogeneous platform is to divide it into smaller portions and pick the processing unit that fits better to run each portion. Hence, system resources can cooperate together to execute the application. Finally, and probably the most important, the requirements derived from the first and second conclusions can be implemented inside runtime frameworks, so the complexity of programming heterogeneous architectures is completely hidden from the programmer point of view.

Acknowledgements

This work has been supported by the following institutions: the European Commission through the ENCORE project (FP7-248647), the TERAFLUX project (FP7-249013), the TEXT project (FP7-261580), the HiPEAC-2 Network of Excellence (FP7/ICT 217068) and the HiPEAC-3 Network of Excellence (FP7-ICT 287759), the support of the Spanish Ministry of Education (CSD2007-00050 and FPU program), the projects of Computación de Altas Prestaciones V and VI (TIN2007-60625, TIN2012-34557), the Generalitat de Catalunya (2009-SGR-980, 2014-SGR-1051), the Intel-BSC Exascale Lab and IBM/BSC Exascale Initiative collaboration and the Severo Ochoa recognition SEV-2011-0067.

I would also like to thank the Universitat Jaume I from Castelló and KAUST IT Research Computing for granting access to their machines.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context and Motivation

Heterogeneous high-performance computers have become a key evolution of regular homogeneous CPU systems due to their computing power. The addition of specialized processing units (accelerators) to regular CPU systems has led a tremendous revolution in the High Performance Computing (HPC) world: on the one hand, the ratio between performance and energy consumption, also known as *performance per Watt*, has significantly increased. On the other hand, vendors offer such kind of devices at very competitive prices, so the ratio between performance and hardware prices, also known as *performance per Euro/Dollar*, has considerably raised as well. The main examples of these recent hardware accelerators include General Purpose GPUs (GPGPUs) [1, 2], the Intel Xeon Phi [3] or FPGAs [4].

Table 1.1 compares the peak performance (single and double precision), power consumption and cost of three different accelerators from nVIDIA, AMD and Intel and one general purpose CPU from Intel respectively [5, 6, 7, 8, 9, 10]. The general purpose CPU has clearly the lowest performance per Watt and performance per Dollar ratios: while its performance is two orders of magnitude lower than the accelerators, it consumes around half the power of accelerators. Moreover, it is the most expensive architecture.

The current TOP500 list (from June 2015) [11] reflects the popularity of heterogeneous com-

Architecture	**Peak Perf. (GFlop/s)**		**Power Consumption (W)**	**Cost ($)**[a]
	SP	DP		
nVIDIA GTX 980 Ti	6144	192	250	806
AMD FirePro S9150	5070	2530	235	3323
Intel Xeon Phi 7120P	2416	1208	300	3883
Intel Xeon E7-8770	96	48	130	5402

Table 1.1: Comparison between different architecture product characteristics

[a] Reference prices were extracted from US Amazon website as of July 2015.

puters, as four of the top 10 machines have either GPGPUs or Xeon Phi coprocessors, the top 2 being heterogeneous computers. Moreover, this trend has held for the last two years. We can then expect that future many-core chips will be heterogeneous (different type of processors, including accelerators) and may be organized in a certain hierarchy (for example, organized in clusters of cores, possibly with associated local memory).

Heterogeneity adds adaptability to a machine, as different processing units of a system may offer better capabilities for specific computations. For example, compute-intensive algorithms fit perfectly on GPGPUs, as they offer a peak performance that can be up to 2 orders of magnitude greater compared to a normal CPU. To make the most of heterogeneous systems, applications should be split into different portions depending on their computational characteristics and requirements. Then, ideally, each portion, should be executed on the processing unit that will run it in the most efficient way.

However, heterogeneity makes the programming task more difficult, especially for programmers that want to fully exploit machine resources. Firstly, some accelerators, like GPGPUs, may not be able to run regular CPU code; others, like the Intel Xeon Phi, are able to run CPU code directly, but it may not exploit its massively-parallel hardware. So, in any case, applications targeting traditional CPU architectures may have to be redesigned and rewritten. Secondly, even in single-node systems, accelerators may have their own separated memory space with limited capacity and access restrictions. From the programmer's point of view, CPUs usually have a single memory space with several transparent mechanisms that speed-up memory accesses, like the different levels of hardware caches. But accelerators, like GPGPUs, expose their memory hierarchy to programmers and leave them the decision of where to put each piece of data as well as the responsibility of moving data from one memory space to another. This forces programmers to include additional code to manage data movements across different memory spaces and to keep data coherency. Finally, distributing the execution of an application among all the computing resources of a system is a non-trivial task and heterogeneity makes it even more complex: synchronizations become particularly complicated if we want to split the computation between multiple devices and still want to get optimal performance. Therefore, code maintenance and performance portability become more complex (and expensive) as source codes must be adapted to fit new architectures. In other words, heterogeneity has raised the programming wall up to an unaffordable level for programmers.

Several proposals have arisen in the last years to program accelerators, the most important being CUDA [12], which targets NVIDIA GPUs, and OpenCL [13], which works with Intel Xeon Phi cards and GPGPUs as well. However, none of them addresses the aforementioned challenges, since they both expose the underlying hardware to the programmer and only offer a resource management API. Consequently, programming models should be able to hide heterogeneity and hierarchy in a way that applications are unaware of the underlying hardware and that can dynamically adapt to it. Moreover, this would allow programmers focus on their application development and forget about the management of available resources, data movements and synchronizations.

The code in Figure 1.1 implements the scale function in CUDA: it multiplies vector A by the scalar sc. The figure illustrates the additional actions that programmers must add in their source codes in order to offload a computation to a GPGPU in CUDA: first, allocate and transfer memory to the device. Then, offload the computation and synchronize with the

```
__global__ void scale_kernel (int n, double *A, double sc)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;

  if (idx < n)
    A[idx] = A[idx] * sc;
}

void scale (int n, double *A, double sc)
{
  // Allocate device memory
  double *dA;
  cudaMalloc(&dA, n * sizeof(double));

  // Transfer data to device memory
  cudaMemcpy(&dA, &A, n * sizeof(double), cudaMemcpyHostToDevice);

  // Offload computation
  dim3 dimBlock(256);
  dim3 dimGrid ((n / dimBlock.x) + 1);
  scale_kernel<<< dimGrid, dimBlock >>>(n, dA, sc);

  // Synchronize with device
  cudaDeviceSynchronize();

  // Copy result back to host
  cudaMemcpy(&A, &dA, n * sizeof(double), cudaMemcpyDeviceToHost);

  // Free device memory
  cudaFree(dA);
}
```

Figure 1.1: A simple CUDA implementation of scale function

device. In addition, the device kernel implementation (*scale_kernel*) must be provided by the programmer. Finally, copy the result back to the host and free all the allocated resources. This source code is not optimized and its only objective is to show all the additional steps that must be done to offload a computation to a GPGPU. In order to optimize this code, these operations should be split into smaller parts, each one processing smaller pieces of the vector, so that computations and data transfers can be overlapped. Writing a code that splits the computation across multiple devices would even be more complex.

1.2 Programming Challenges in Heterogeneous Systems: One code does not fit all

In general, there is not a single piece of code that fits all the existing hardware architectures, and even if we find that code, it will not be the best (in terms of performance, energy consumption, etc.) for all of them. Thus, it is not unusual to find different ways of implementing the same algorithm. As an example, there are uncountable versions of the matrix multiply algorithm. Figure 1.2 shows a simple CPU implementation of a tiled matrix multiply algorithm that could run in different architectures. However, this is not the optimal version for any of them.

There are many libraries that implement highly optimized versions of this algorithm for CPUs, like BLAS [14], LAPACK [15] or MKL [16]. The matrix multiply implementation found in these libraries has vectorized code to fully exploit the CPU hardware capabilities and is usually optimized for particular matrix sizes (e.g. power-of-two matrix sizes or square matrices).

```c
void dgemm (int m, int l, int n, double *tileA, double *tileB, double *tileC)
{
  for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
      for (int k = 0; k < l; k++)
        tileC[i*n+j] += tileA[i*l+k] * tileB[k*n+j];
}

void matmul (int m, int l, int n, double **A, double **B, double **C, int tm, int tl, int tn)
{
  for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
      for (int k = 0; k < l; k++)
        dgemm(tm, tl, tn, A[i*l+k], B[k*n+j], C[i*n+j]);
}
```

Figure 1.2: A simple C implementation of matrix multiply

However, none of the aforementioned CPU libraries will work on GPGPUs. Instead, there are specific GPU-compatible implementations that can be found in libraries like CUBLAS [17] or MAGMA [18]. Indeed, these implementations are carefully designed to fully exploit GPU capabilities and massively parallel hardware. Thus, these codes are highly parallelized, take into account how memory accesses are performed (as certain sequences of memory accesses give higher memory bandwidths), make use of GPU registers and shared memory to reduce memory latency, etc. Also, these codes are usually optimized for certain matrix sizes and, depending on that, different kernel configurations (thread grid size and block size) are selected as well. In addition, some libraries, like CUBLAS, have several internal implementations optimized for a particular GPGPU hardware family, as different hardware families may present different characteristics and require different optimizations.

For example, the code in Figure 1.3 illustrates a simple CUDA implementation of matrix multiply. In contrast with the scale function presented before, there is no need to provide the kernel code in this case, as the application calls CUBLAS to offload the computation to the GPGPU. However, CUBLAS requires additional initialization and clean-up operations that must be done by the programmer. In this case, implementing a tiled version of this algorithm would make it possible to overlap operations or even split the computation across several GPGPUs. However, this would introduce much more complexity in terms of programmability.

To make things even more complicated, the Xeon Phi cards present a combination of CPU and GPGPU properties: on the one hand, Xeon Phi cards are built with regular CPUs, so vectorization is a key factor for any code to achieve good performance. On the other hand, it offers a massively parallel hardware where hundreds of threads can run simultaneously. Then, all these hardware properties must be taken into account to write optimized codes. In this case, the native MKL library for Xeon Phi [19] provides a matrix multiply implementation especially designed for this type of architecture.

The source code in Figure 1.4 shows the same simple matrix multiply algorithm targeting the Xeon Phi architecture. The hStreams [20] library provided by Intel is used to manage the offloading and call the native MKL library to perform the computation on the device. Like the CUDA version, this code is not optimized and a more complex tiled structure would be needed to make it more efficient. These codes reflect the diversity and complexity that programmers must face when writing applications for heterogeneous systems.

```
void matmul (int m, int l, int n, double *A, double *B, double *C)
{
  // Allocate device memory
  double *dA, *dB, *dC;
  cudaMalloc(&dA, m * l * sizeof(double));
  cudaMalloc(&dB, l * n * sizeof(double));
  cudaMalloc(&dC, m * n * sizeof(double));

  // Initialization needed by device libraries
  cublasHandle_t handle;
  cublasCreate(&handle);

  // Transfer data to device memory
  cudaMemcpy(&dA, &A, m * l * sizeof(double), cudaMemcpyHostToDevice);
  cudaMemcpy(&dB, &B, l * n * sizeof(double), cudaMemcpyHostToDevice);
  cudaMemcpy(&dC, &C, m * n * sizeof(double), cudaMemcpyHostToDevice);

  // Offload computation
  cublasDgemm(handle, CUBLAS_OP_T, CUBLAS_OP_T, m, n, l, &alpha,
          dA, l, dB, n, &alpha, dC, m);

  // Synchronize with device
  cudaDeviceSynchronize();

  // Copy result back to host
  cudaMemcpy(&C, &dC, m * n * sizeof(double), cudaMemcpyDeviceToHost);

  // Free device memory and clean-up device libraries
  cudaFree(dA);
  cudaFree(dB);
  cudaFree(dC);
  cublasDestroy(handle);
}
```

Figure 1.3: A simple CUDA implementation of matrix multiply calling CUBLAS library

```
void matmul (int m, int l, int n, double *A, double *B, double *C)
{
  // Initialize hStreams library and allocate device memory
  hStreams_app_init(1, 1);
  hStreams_app_create_buf(A, m * l * sizeof(double));
  hStreams_app_create_buf(B, l * n * sizeof(double));
  hStreams_app_create_buf(C, m * n * sizeof(double));

  // All operations are asynchronous, but use the same stream,
  // so hStreams will control data dependences between them
  int stream = 1;

  // Transfer data to device memory
  hStreams_app_xfer_memory(A, A, m * l * sizeof(double), stream, HSTR_SRC_TO_SINK, NULL);
  hStreams_app_xfer_memory(B, B, l * n * sizeof(double), stream, HSTR_SRC_TO_SINK, NULL);
  hStreams_app_xfer_memory(C, C, m * n * sizeof(double), stream, HSTR_SRC_TO_SINK, NULL);

  // Offload computation, native MKL will be invoked on device
  hStreams_app_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, n, l,
          1.0, A, m, B, l, 1.0, C, m, stream, NULL);

  // Copy result back to host
  hStreams_app_xfer_memory(C, C, m * n * sizeof(double), stream, HSTR_SINK_TO_SRC, NULL);

  // Synchronize with device
  hStreams_app_thread_sync();

  // Free device memory and finalize hStreams library
  hStreams_DeAlloc(A);
  hStreams_DeAlloc(B);
  hStreams_DeAlloc(C);
  hStreams_app_fini();
}
```

Figure 1.4: A simple hStreams implementation of matrix multiply for Intel Xeon Phi

Despite these software development problems, accelerators have become very popular thanks to its capabilities: a small amount of them can give the same computational power as a supercomputer at a considerably lower price. Unfortunately, they are not the universal solution to accelerate existing applications, as they can perform very well at compute-intensive algorithms, but we can find a large set of applications where accelerators get no speed-up, or even perform worse than CPUs.

Nowadays, it is the programmer responsibility to consider whether their code (or a part of it) can benefit from accelerator capabilities and to add all the necessary operations to perform such offloading. Nevertheless, there are many research lines opened in software systems for heterogeneous computers. Hopefully, these software systems will become smart enough in the future to make all these decisions automatically and transparently to help programmers. The work developed in the context of this thesis is an example and other research proposals are illustrated in Chapter 3 as well. For example, the OpenMP and OpenACC directive-based programming models are two high-level proposals that seek lowering the programming wall by offering code portability across different architectures.

1.3 Contributions

The main goal of this PhD thesis is to design the appropriate scheduling techniques and resource management for heterogeneous systems. The objective is to include them into a runtime system, so that they are completely automatic and transparent to programmers. In order to prove the validity of this work and increase its impact on the scientific community, the research has been focused on the two most common accelerators that exist nowadays: GPGPUs and Intel Xeon Phi.

In recent years, task programming has become popular and fits perfectly well in heterogeneous programming, as each task can be seen as the sequential piece of code that is run by one of the machine resources. However, developing a new programming model from scratch is out of the scope of this thesis. OmpSs is an OpenMP-like task-based programming model. It has been carefully designed in a modular way and can be extended easily with new features. OmpSs offers flexibility to programmers and adaptability to current and future hardware architectures. For these reasons, OmpSs was chosen as the basis of this work and all thesis contributions have been developed and tested on top of it.

The contributions of this thesis can be divided into two main areas described below:

❏ **Accelerator support and management**: we demonstrate that it is possible to hide the difficulties of accelerator programming from the programming model side. In this sense, we have designed an approach capable of performing all the accelerator-related management (like execution offloading, data movements or synchronizations) that is completely transparent and needs no additional effort from the programmer. This approach mainly targets task-based programming frameworks and adds the specific support needed to offload task execution to GPGPU and Xeon Phi accelerators.

❏ **Scheduling techniques for heterogeneous architectures**: we provide a set of scheduling techniques especially designed for heterogeneous architectures. Each technique focuses on a different objective (like minimizing the amount of data transfers, prioritizing the execution of critical paths, or maximizing the resource utilization) and they can also be combined together. We prove that our scheduling techniques can dynamically decide the best task execution order and distribution for a particular objective.

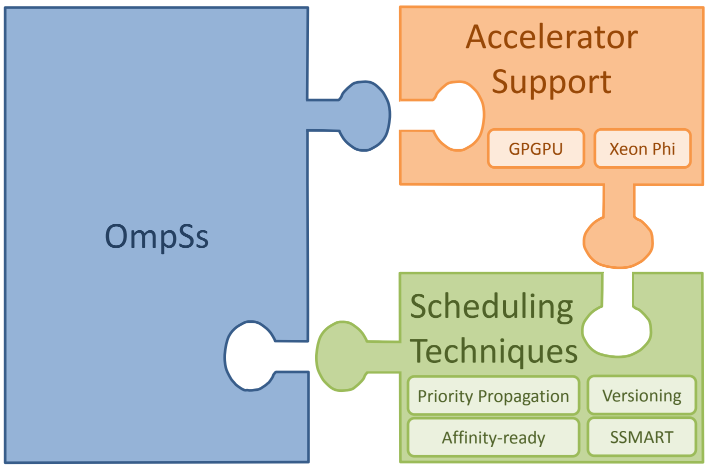


Figure 1.5: Illustration of thesis contributions

Figure 1.5 pictures the thesis contributions and how they are integrated in the OmpSs framework. The accelerator support and scheduling techniques have been added into the OmpSs framework and can be combined to execute OmpSs applications.

The publications that support this thesis are listed below in chronological order.

- [21] Javier Bueno, **Judit Planas**, Alejandro Duran, Xavier Martorell, Eduard Ayguadé, Rosa M. Badia and Jesús Labarta. *Productive Programming of GPU Clusters with OmpSs.* Paper presentation, 26th IEEE International Parallel and Distributed Processing Symposium. May 21-25, 2012. Shangai, China.

- [22] **Judit Planas**, Rosa M. Badia, Eduard Ayguadé and Jesús Labarta. *Self-Adaptive OmpSs Tasks in Heterogeneous Environments.* Paper presentation, In proc. of 27th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2013). Boston, Massachusetts, USA. May 2013.

- [23] **Judit Planas**, Rosa M. Badia, Eduard Ayguade, and Jesus Labarta. *AMA: Asynchronous Management of Accelerators for Task-based Programming Models.* Paper presentation, International Conference on Computational Science (ICCS 2015), Volume 51, Pages 130-139. Reykjavik, Iceland. June 2015.

- **Judit Planas**, Rosa M. Badia, Eduard Ayguadé and Jesús Labarta. *SSMART: Smart Scheduling of Multi-ARchitecture Tasks on Heterogeneous Systems. Submitted to Second Workshop on Accelerator Programming using Directives (WACCPD).* 2015.

The contributions of this thesis have also been published as part of a collective group work. The related publications are listed below in chronological order.

- ❏ [24] Eduard Ayguadé, Rosa M. Badia, Pieter Bellens, Daniel Cabrera, Alejandro Duran, Marc Gonzalez, Francisco Igual, Daniel Jimenez-Gonzalez, Jesús Labarta, Luis Martinell, Xavier Martorell, Rafael Mayo, Jose M. Perez, **Judit Planas** and Enrique S. Quintana-Ortí. *Extending OpenMP to Survive the Heterogeneous Multi-core Era.* Journal publication, International Journal of Parallel Programming, Vol. 38, No. 5-6, 440-459. June 2010.

- ❏ [25] R. Ferrer, **J. Planas**, P. Bellens, A. Duran, M. Gonzalez, X. Martorell, R. Badia, E. Ayguadé and J. Labarta. *Optimizing the Exploitation of Multicore Processors and GPUs with OpenMP and OpenCL.* Paper presentation, In proceedings of the 23rd International Workshop on Languages and Compilers for Parallel Computing. October, 2010. Houston, Texas, USA.

- ❏ [26] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell and **Judit Planas**. *OmpSs: A proposal for programming heterogeneous multi-core architectures.* Journal publication, Parallel Processing Letter, Volume 21, Issue 2, pp. 173 - 193. June, 2011.

- ❏ [27] Eduard Ayguade, Rosa M. Badia, Pieter Bellens, Javier Bueno, Alex Duran, Yoav Etsion, Montse Farreras, Roger Ferrer, Jesus Labarta, Vladimir Marjanovic, Lluis Martinell, Xavier Martorell, Josep M. Perez, **Judit Planas**, Alex Ramirez, Xavier Teruel, Ioanna Tsalouchidou and Mateo Valero. *Hybrid/Heterogeneous Programming With OmpSs And Its Software/Hardware Implications.* Book chapter, In Programming Multicore and Many-core Computing Systems (Wiley Series on Parallel and Distributed Computing), ed. John Wiley & Sons, Inc. January 2012.

- ❏ Chris J. Newburn, Gaurav Bansal, **Judit Planas**, Alejandro Duran, Paulo Souza, Leonardo Borges and Jesus Labarta. *Heterogeneous Streaming. Under submission process.*

The next sections describe the contributions in more detail and link them with the publications.

1.3.1 Accelerator Support and Management

This thesis contributes with the design and implementation of an efficient way to give support and manage task execution on accelerators. Our design is based on the asynchronous property that most accelerators present and focuses on minimizing the amount of time spent in host-device synchronization. As explained before, it mainly focuses task-based programming frameworks. With our contribution, we add all the necessary support to the target framework to be able to offload tasks to a given accelerator. This support includes (i) runtime operations such as device initialization and configuration (if needed), data movements, host-device synchronization and task offloading and (ii) compiler capabilities to apply the appropriate source code transformations to enable task offloading to accelerators. This model has been presented in several publications [24], [26] and [27].

Figure 1.6 illustrates the OmpSs implementation of the simple matrix multiply code presented before. By just inserting the `target`, `task` and `taskwait` directives, OmpSs will offload the computation to a GPGPU and will handle all the necessary memory movements. In this

```
 1 #pragma omp target device (cuda) copy_deps
 2 #pragma omp task inout([m][n]tileC) in([m][l]tileA, [l][n]tileB)
 3 void matmul_kernel (int m, int l, int n, double *A, double *B, double *C)
 4 {
 5     double alpha = 1.0;
 6
 7     // Get the appropriate execution stream and CUBLAS context from OmpSs runtime
 8     cublasHandle_t handle = nanos_get_cublas_handle();
 9     cudaStream_t stream = nanos_get_kernel_execution_stream();
10     cublasSetStream(handle, stream);
11
12     cublasDgemm(handle, CUBLAS_OP_T, CUBLAS_OP_T, m, n, l, &alpha,
13             A, m, B, l, &alpha, C, m);
14 }
15
16 void matmul (int m, int l, int n, double *A, double *B, double *C)
17 {
18   // No need to allocate nor transfer device memory
19   // CUBLAS library is initialized inside OmpSs runtime
20
21   // Offload computation
22   matmul_kernel(m, l, n, A, B, C);
23
24   // Synchronize with device
25   // This synchronization also brings output data back to the host
26   #pragma omp taskwait
27
28   // No need to free device memory nor clean-up CUBLAS library
29 }
```

Figure 1.6: Simple OmpSs matrix multiply code using the GPGPU to offload the computation

case, since the CUBLAS library is used, the code in lines 8-10 is required by the runtime to properly manage the offloading. This CUBLAS particularity is explained and justified in detail in Section 4.1.2.5.

Regarding the runtime contribution, two different prototypes where explored: first, a semi-asynchronous approach and, second, a fully asynchronous approach.

1.3.1.1 Semi-asynchronous Approach

The semi-asynchronous approach interleaves asynchronous periods with fully blocking synchronized points because the existing technologies, by the time it was implemented, did not offer any possibility for a fully asynchronous host-device communication behavior.

Having blocking synchronizations negatively affects performance, because this means that the CPU has to block at some points and waste its time while it is waiting for the device to finish its operations (data transfers and computations). But this was the only way to check for the state of operations that were issued to the device. So, we found a compromise between the number of blocking synchronization points and the level of control the host had over the device.

The semi-asynchronous approach was only implemented and tested on GPGPUs devices because it was the most famous and almost the only type of accelerators that existed by that time. This work was published as a conference paper [21].

1.3.1.2 Fully asynchronous Approach

Although the semi-asynchronous approach was good enough for its time, as the technology evolved, accelerators offered new features that could clearly improve our first approach. Thus, a second prototype, called AMA (Asynchronous Management of Accelerators), was designed with the property of being completely asynchronous.

AMA is able to completely remove the previous blocking synchronizations and establish and event-driven communication between host and device. This approach clearly enhances the existing design because it completely decouples the operations done on both host and device sides. The main idea behind this design is to use the CPU to do some other useful work while it is waiting for the device to finish its computations.

We have proved the validity of the fully asynchronous approach with two of today's most famous accelerators: GPGPUs and Intel Xeon Phi. This model and its GPGPU implementation were presented as a conference paper [23]. The Intel Xeon Phi related work is under publication process.

1.3.1.3 Accelerator Compiler Support

In order to make the accelerator programming task easier, the accelerator compiler support tries to minimize as much as possible the source code differences between calling a regular CPU task or an accelerator task. For example, in the case of GPGPUs, the compiler inserts calls to the runtime that will decide at run time which execution stream is used for each kernel, being this completely transparent from the programmer point of view. This work was published as a conference paper [25].

1.3.2 Scheduling Techniques for Heterogeneous Architectures

The second thesis contribution is a set of scheduling techniques designed and optimized for heterogeneous architectures. Also targeting task-based programming frameworks, each of our scheduling technique focuses on a specific objective and finds the best task execution order and distribution to accomplish such objective.

The set of scheduling techniques focuses on the following objectives: prioritize the execution of the critical path of an application (task priority propagation), minimize the amount of data transferred (affinity-ready scheduler), maximize system's resource utilization (versioning scheduler) and a combination of maximizing resource utilization while minimizing the amount of data transfers (SSMART scheduler).

1.3.2.1 Task Priority Propagation

Assuming that we have a task graph where tasks that belong to the critical path[1] are assigned higher priorities than those outside the critical path, the task priority propagation mechanism helps finding such tasks with higher priority in advance.

[1] The critical path of an application task graph is the sequence of tasks determining the minimum time needed to execute the application.

We do not consider this mechanism a scheduling technique itself, but it can be combined with any other scheduling technique to help it improve the order in which tasks are being run. This work was published in a conference paper [23].

1.3.2.2 Affinity-ready Scheduler

This scheduling technique checks for task data locality and assigns each task to the processing unit that needs less data to be transferred to run the given task. It is especially designed for accelerators by carefully accounting the number of total bytes that need to be transferred for each task.

In addition, it can be combined with the task priority mechanism to enhance the order in which tasks are selected. This work was published in a conference paper [23].

1.3.2.3 Versioning Scheduler

The versioning scheduling technique contributes in two aspects of the programming framework. First, it adds the ability to join separate pieces of code (i.e. new or alternative task implementations) to the original application without having to modify it. And second, proposes a new scheduling strategy to evaluate these new added pieces. The aim of this contribution is performance portability at low-cost maintenance: programmers can add different task implementations of the same algorithm (that can also target different architectures) and let the runtime dynamically explore and decide which implementation is chosen each time a task is called.

The main objective of the scheduling strategy is to increase application performance by maximizing resource utilization: task[2] performance is monitored on each processing unit and a look-ahead scheduling is done to find the best distribution of future tasks on system resources. Since all system resources cooperate to run the application, its performance can be potentially increased. This work was presented as conference paper [22].

1.3.2.4 SSMART Scheduler

The SSMART scheduler completes the set of scheduling techniques by combining and improving some of the aforementioned scheduling objectives. On the one hand, it enhances the versioning scheduler in several aspects and, on the other hand, combines both versioning and affinity-ready objectives.

In short, SSMART extends the versioning scheduler characteristics in the following aspects: first, SSMART takes into account data locality to decide which processing unit is the most suitable to run a task as well as the estimated time needed for data transfers. Second, for each pair task-processing unit, SSMART creates a rank of suitability (how good is a processing unit to run the given task, taking into account its workload and task data locality). Then, if needed, the scheduler is able to decide the most suitable device within a subset of the processing units of the whole system. Third, SSMART allows stealing tasks that were already assigned to a certain processing unit. Fourth, it supports task priority propagation. Finally,

[2] Only those tasks that have more than one implementation are taken into account in the monitoring and scheduling process. The rest of the tasks are scheduled following a breadth-first strategy.

versioning only profiles tasks with more than one implementation, while SSMART profiles all tasks of the application. This work is under publication process.

1.4 Thesis Organization

This thesis document is structured in the following way: this chapter, Chapter 1, is the introduction and describes the motivations, the context and the contributions of this thesis. Chapter 2 gives an overview of OmpSs, the programming model and framework used as the basis to develop the contributions of the thesis. It also includes a description of the applications that have been used to evaluate the implementations. The following chapter, Chapter 3, summarizes the state of the art with respect to each thesis contribution. Then, Chapters 4 and 5 give a detailed description of the contributions and present their evaluation. The conclusions and future work of this thesis are discussed in Chapter 6. Finally, the reader can find the bibliography at the end of this document.

In addition, as a complement of the preceding chapters, Appendix A shows the specific OmpSs runtime options for GPGPU and Xeon Phi devices and Appendix B compares the source codes of an OmpSs version of a tiled matrix multiply algorithm with sequential C, CUDA, OpenMP and hStreams versions.

Chapter 2

Development Environment

This chapter introduces the OmpSs programming model and describes the applications that have been used in the context of this thesis.

2.1 OmpSs programming model

The OmpSs programming model [24] defines a single task-based programming model for homogeneous and heterogeneous architectures and is open to support new architectures that may appear in the future.

OmpSs combines ideas from OpenMP [28] and StarSs [29]: on the one hand, it enhances OpenMP with support for irregular and asynchronous parallelism and heterogeneous architectures and, on the other hand, it incorporates StarSs dependence support [30] and data-flow concepts that allow the framework to automatically move data as necessary and perform different kinds of optimizations. OmpSs is currently able to run applications on clusters of nodes that combine shared memory processors (SMPs) and other external devices, for example, FPGA, GPU and Xeon Phi [21, 23, 31], being the last two contributions of this thesis.

2.1.1 Execution Model

OmpSs uses a thread-pool execution model instead of the traditional OpenMP fork-join model. There is a master thread that starts the execution of the application and several other threads that cooperate executing the work it creates from worksharing or `task` constructs. Therefore, unlike OpenMP, there is no need for a `parallel` region. Nesting of constructs allows other threads to generate work as well.

An application's inherent parallelism is exploited from `task` constructs at runtime: a data-dependency graph is dynamically built with the information extracted at compile time from parameter directionality clauses (`in`, `out` and `inout`). This task graph construction is essential to keep application's data coherence and correctness. Then, only ready tasks (i.e. tasks that do not depend on other tasks) can be run in parallel.

2.1.2 Memory Model

From the application point of view, there is a single address space. However, internally, OmpSs assumes that multiple address spaces may exist. Then, data can be shared between these address spaces and may reside in memory locations that are not directly accessible from some of the computational resources. Thus, all parallel code can only safely access private and shared data that has been marked explicitly with OmpSs extended syntax. This assumption is true even for SMP machines as the implementation may reallocate shared data to improve memory accesses in, for example, NUMA systems [32].

The runtime takes care of where data resides by means of a directory structure. The architecture support manages data transfers between memory spaces as tasks consume or produce them. Data can be replicated on different memory spaces and coherency is transparently managed by the runtime.

2.1.3 OmpSs Syntax Extensions

OmpSs supports OpenMP directives in general, but, additionally, it defines several extensions to OpenMP's syntax that are explained below:

- **Dependency synchronization**: OmpSs integrates the StarSs dependence support. It allows annotating tasks with three clauses: `in`, `out` and `inout`. They allow expressing, respectively, that a given task depends on some data produced before, that it will produce some data, or both. The clause allows specifying arrays, pointers and pointed data. Data addresses and sizes do not need to be constant at compile time since they are computed at execution time. In addition, the `taskwait` construct is extended with the `on` clause, which allows the encountering task to block only until the set of data specified in this clause is produced. `taskwait` has been extended as well with the `noflush` clause which allows synchronizing tasks without flushing all the data on remote devices.

  Dependence clauses have been recently introduced in OpenMP 4.0: the OpenMP `depend` clause is now very similar to the OmpSs `in`, `out` and `inout` clauses. However, OmpSs is able to detect dependences between non-contiguous or strided regions whereas OpenMP uses the initial address of a region to detect task dependencies. Therefore, partially overlapping or strided regions cannot be detected [33, 34].

- **The target construct**: This extension was introduced to support heterogeneity and data motion and it is represented by the `target` construct [35]. It can be applied to tasks and accepts the following clauses:

  - `device`: It specifies which devices can run the associated code (e.g., fpga, gpu, smp). SMP device (for CPUs) is assumed by default. The construct `target device` can also be applied to other functions that are not necessarily tasks.
  - `copy_in`: It specifies that some data must be accessible to the task when running. This may imply a data transfer between memory spaces.
  - `copy_out`: It specifies that some data that was accessible to the task when running will be the only valid version when the task finishes its execution.
  - `copy_inout`: This clause is a combination of `copy_in` and `copy_out`.

- `copy_deps`: It specifies that any task's dependence clause will also have copy semantics (i.e., `in` will be also considered `copy_in`, `out` will be also `copy_out` and `inout` will be also `copy_inout`). To make sure that data that were moved to a device are valid again in the host, SMP tasks (for CPUs) must also use the `copy` clauses or appear after an either implicit or explicit OpenMP `flush`.
- `implements`: This clause is used to specify that the annotated task is an implementation of another task and has been developed in the context of this thesis.

Since OpenMP 4.0, the `target` construct is accepted in OpenMP as well. But the semantics are very different: while the OpenMP construct forces the programmer to specify which data movements must be performed, OmpSs offers the possibility to deduct the necessary data transfers from the data-dependency clauses (`copy_deps` clause). Moreover, The OpenMP `device` receives an integer parameter that identifies the physical unit where the task will be run. In contrast, OmpSs `device` clause is used to specify the architecture type of the task; the runtime will transparently decide which physical unit (of the appropriate architecture type) executes the task. This approach is more flexible and the same source code can run in different machine configurations regarding the number and type of available processing units. In addition, OmpSs is not able to generate device code, so the appropriate kernel code (e.g. in CUDA, OpenCL, etc.) is required.

The different copy clauses do not necessarily imply a copy before and after the execution of each task. This allows the runtime to take advantage of devices with access to the shared memory or implement different data caching and prefetching techniques without the user needing to modify their code.

The OmpSs framework is distributed with two different components: the Nanos++ runtime library and the Mercurium source-to-source compiler, both explained below.

2.1.4 Nanos++ Library

Nanos++ [36] is an extensible runtime library that supports the OmpSs programming model. Its responsibility is to schedule and execute *parallel tasks* as specified by the compiler, based on constraints specified by the user: order, coherence, etc.

Most of the runtime components are independent from the actual target architectures, so general runtime characteristics are explained in this chapter and the specific architecture support developed in the context of this thesis is described in Chapter 4. As of today, Nanos++ supports the following *conceptual* architectures:

- **smp**: targets general purpose CPUs.
- **smp-numa**: used in Non-Unified Memory Access (NUMA) systems with general purpose CPUs.
- **gpu** [37]: offloads tasks in CUDA-capable GPGPUs.

- **opencl**: offloads tasks to any architecture that supports OpenCL (currently CPUs, GPGPUs and Xeon Phi).
- **hstreams**: offloads tasks to any architecture supported by the hStreams library (currently Xeon Phi, general purpose CPUs will be supported in the future as well).
- **fpga**: offload tasks to FPGA devices.
- **cluster** [38]: distributes tasks among the different nodes of a cluster system and can be combined with other conceptual architectures that the hardware system may offer.
- **mpi**: distributes tasks among the different nodes of a cluster system using the Message Passing Interface (MPI) [39] library.
- **tasksim** [40]: used in a simulated architecture.

The following sections give an overview of the most important independent mechanisms of the library that serve as glue between the different architectures.

2.1.4.1 Nanos++ Independent Layer

When a piece of code annotated as a `task` is reached, the runtime creates a new task. The data environment of the task is captured from the function arguments or scope variables and is used to dynamically build a task data-dependency graph to ensure program's correctness. When the task data dependencies are satisfied (usually by the completion of its predecessor tasks), the task becomes ready and can be run.

The task life can be divided into five stages (from its creation to its completion), described in chronological order:

- **Instantiation**: OmpSs creates the task and all its related data structures. The dependency support computes task data dependencies and adds a new node to the task graph representing this task. The appropriate connections between the task and its predecessors are created.
- **Ready**: task's data dependencies are satisfied, typically as a result of the completion of the predecessor tasks. Task scheduling usually occurs in this stage[1].
- **Active**: this stage includes all the operations needed once the task has been scheduled to a system's processing unit and before the task can be run. The coherence layer is invoked in this stage to ensure that all necessary data is available in the appropriate memory space. Then, if needed, data allocations and input data transfers are issued.
- **Run**: the task is executed. The corresponding architecture dependent layer is the responsible for this action and will notify the system when the execution is finished.

[1] Most schedulers decide the unit where the task will be run at this stage, but, for example, the locality-aware scheduler (explained later in this chapter) computes the affinity score and makes this decision in the instantiation stage for performance reasons.

- **Completion**: all the operations needed once the task has been run occur in this stage. The coherence layer is invoked again to process task's output data. Depending on the configuration of this layer, data transfers and data deallocations may occur or not. The dependency support updates the task graph by removing the corresponding task node and its connections. Then, task successors that have no other dependence will become ready.

2.1.4.2 Nanos++ Dependency Support

The runtime maintains a directed acyclic graph where tasks are connected following the dependencies extracted from the directionality clauses specified by the user. Edges between nodes are created for different kinds of dependencies: *read-after-write*, *write-after-read*, *write-after-write*.

The OmpSs model does not allow data dependencies outside the dynamic extent of a given task. This means that only sibling tasks will be connected together. This is particularly important as it allows a hierarchical implementation of the graph for applications with multiple levels of task parallelism.

The dependency support is implemented as a runtime plug-in, so, different behaviors can be implemented and chosen at run time.

2.1.4.3 Nanos++ Task Scheduler

Nanos++ allows changing the scheduler used for each execution and, thus, experimenting with different scheduling strategies. The runtime offers several scheduling policies, explained below. The ones included in this thesis contributions are explained in Chapter 5:

- **Breadth-first**: It follows a simple first in, first out (FIFO) scheduling strategy, but before picking the first task from the ready task queue, it tries to schedule a successor of the task that just finished. The idea behind this is that the successor task will share data and it will likely minimize the number of data transfers between disjoint memory spaces.

- **Locality-aware**: In this strategy, when a new task is submitted, the scheduler computes an affinity score for each system memory space. This score is based on where each piece of data specified by the task clauses is located and also takes into account the size of that data. Then, the score is used to place the task in any of the computing units with direct access to the memory space with the highest affinity. Unlike the other schedulers, affinity-aware makes the decision in the task instantiation stage for performance reasons.

- **NUMA-aware**: The aim of this strategy is to minimize memory latency in NUMA systems. It takes into account the physical location of data and assigns tasks to those processing units that are physically attached to the memory bank where task data resides. Task stealing is allowed as well to reduce load imbalance between processing units.

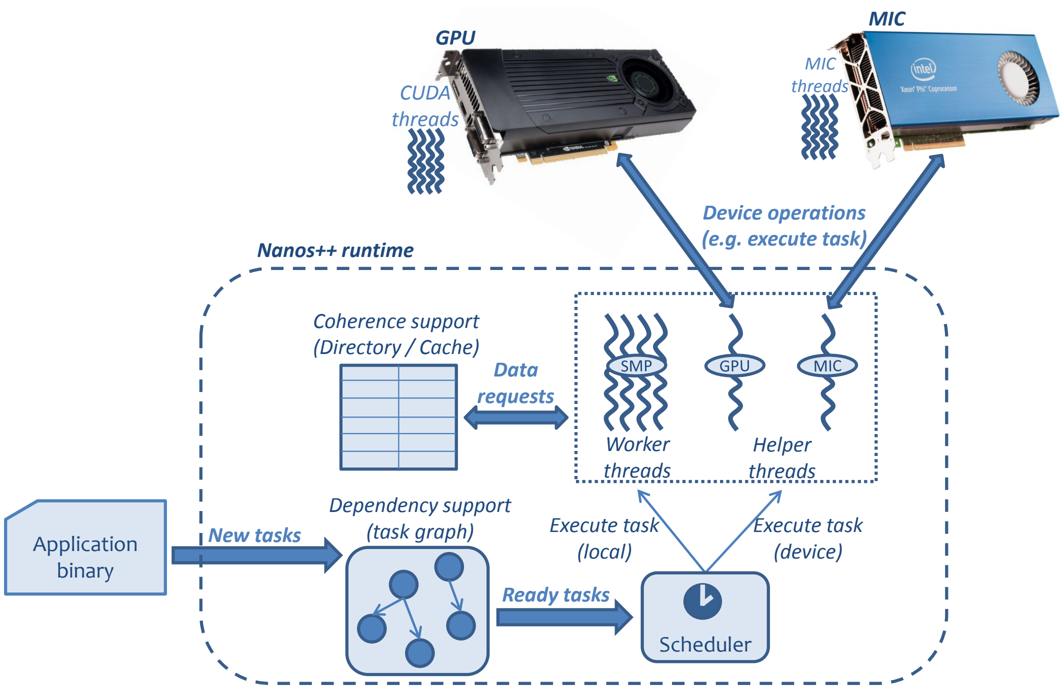


Figure 2.1: Nanos++ runtime components and execution flow

Many schedulers support task priorities[2] to establish a task execution order while preserving data dependencies: tasks with higher priority will be executed earlier.

2.1.4.4 Nanos++ Coherence Support

During the task active stage, just before the task is executed, the coherence support is invoked to ensure that an up-to-date copy of the data is available in the address space where the task is going to run.

A hierarchical directory keeps track of the physical location of data and of the most recent version. In addition, a software *cache* exists for each device that has a separate address space. This *cache* keeps track of which data is in each address space so it allows skipping unnecessary data transfers. The *cache* can work in two different write policies: write-through or write-back, being this last one the default policy.

The coherence layer ultimately invokes the architecture support to perform the necessary data allocations, deallocations and transfers.

It is important to notice that the coherence mechanisms assume program correctness. Applications where the programmer provides tasks that write to the same data simultaneously without specifying proper synchronization (e.g., using the dependency clauses) result in undefined behavior.

[2] *Task priority* and *task priority propagation* are two concepts highly linked, but they should not be confused. Task priority reflects the fact that the task execution order can be modified by giving some priority to certain tasks (usually represented as an integer). Task priority propagation refers to the action of propagating the task priority to its predecessors (usually done by adding its priority to the current predecessor priority).

Figure 2.1 illustrates the different Nanos++ runtime components and its execution flow. The figure shows the runtime components that are loaded in a heterogeneous system with several CPUs, one GPGPU and one Intel Xeon Phi (MIC) card. Thus, several threads are created to run the tasks on the different execution units. Threads that run tasks on CPUs are called *worker threads* and those threads used to manage and offload tasks to accelerators are called *helper threads* and run on the CPU as well. In this case, one helper thread is created for each accelerator. As the OmpSs application binary is executed, task creation code will be reached and, thus, the Nanos++ dependency layer will create a new node in the task dependency graph for each created task. When the task becomes ready, the runtime scheduler component decides which execution unit will run the task. Then, the architecture support for that unit and the coherence layer coordinate the necessary data transfers and run the task. Finally, once the task is run, the dependency and coherence layers are invoked again to update their information.

Thanks to the flexible design and implementation of OmpSs runtime, it is very easy to extend any of its features, like adding a new scheduler or even the support for a new architecture. New features can be added as new plug-ins and later on, when the application is run, the user can decide which plug-ins should be enabled through configuration arguments or environment variables. Thus, the same application can be run several times using, for example, different schedulers, and there is no need to recompile neither the OmpSs runtime nor the application; only the appropriate environment variables or configuration arguments must be set before each execution.

2.1.5 Mercurium Compiler

The compiler [41] plays a relatively minor role on the implementation of the OmpSs model. On one side, the compiler recognizes the constructs and transforms them into calls to the Nanos++ runtime library. The data-flow clauses are transformed into a set of expressions. The evaluation of these expressions at execution time will generate addresses of memory that will be passed to the runtime library to build the task dependency graph.

On the other side, the compiler manages code restructuring for different target devices. When the compiler is about to generate the code for a **`task`** construct it checks if there is a **`target device`** directive. If so, then the appropriate internal representation for the task is passed onto a device-specific *handler* for each non-SMP device [3].

These handlers generate the device-dependent data to be associated with the task. If necessary, they can also generate additional code for different specific devices in separate files. These additional files are reintroduced in the compiler pipeline usually following different compilation profiles that will invoke different backend tools (e.g., in the case of GPU devices, the nVIDIA *nvcc* compiler will be invoked).

Additional files and binaries generated by the compiler are merged together into a single object file that contains additional information about the different subobjects. This way, the compiler maintains the traditional behavior of generating one object file per source file to enable compatibility with other tools, like makefiles. The information is recovered at the linkage step to generate the final binary with all the objects.

[3]The specific GPU device handler has been developed within the context of this thesis and it is explained

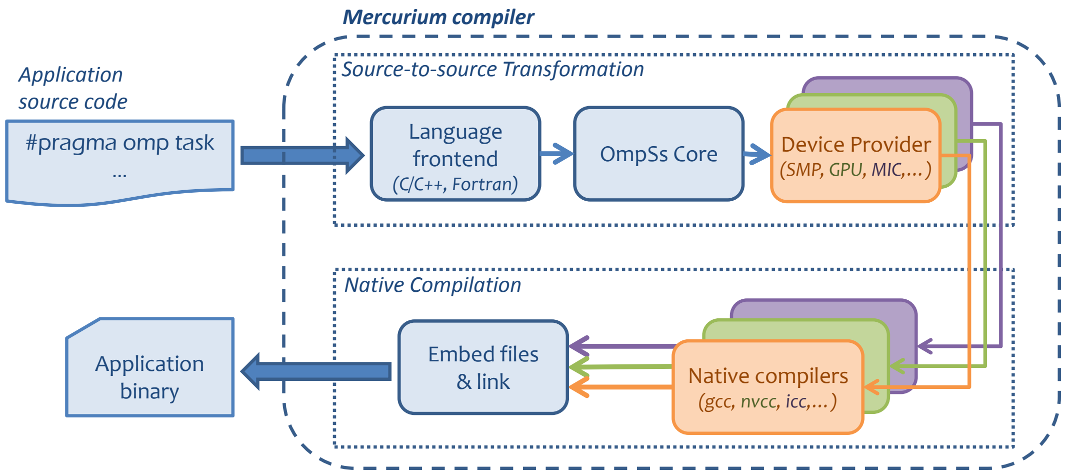


Figure 2.2: Mercurium file compilation flow

```
#pragma omp task inout([ts][ts]tileC) in([ts][ts]tileA, [ts][ts]tileB)
void dgemm_task (double *tileA, double *tileB, double *tileC, int ts)
{
  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, ts, ts, ts, 1.0,
      tileA, ts, tileB, ts, 1.0, tileC, ts);
}

void matmul (int m, int l, int n, double **A, double **B, double **C, int ts)
{
  int i, j, k;
  for (i = 0;i < m; i++)
    for (j = 0; j < n; j++)
      for (k = 0; k < l; k++)
        dgemm_task (A[i*l+k], B[k*n+j], C[i*n+j], ts);

  #pragma omp taskwait
}
```

Figure 2.3: OmpSs tiled matrix multiply calling CBLAS

Figure 2.2 shows the file compilation flow of an OmpSs application. First, Mercurium does a source-to-source transformation of the application source code annotated with directives by replacing the constructs with calls to the Nanos++ runtime. Then, host and device codes are split into different files and are processed by the corresponding device provider component. Finally, each piece of code is compiled with its appropriate native compiler and all the resulting object files are linked together to generate the OmpSs application binary.

2.1.6 OmpSs Application Example

This section illustrates a tiled matrix multiply algorithm written in OmpSs as a didactic example. The algorithm takes two input matrices, A and B of $m \times l$ and $l \times n$ elements respectively, and computes its product in a resulting matrix, C of $m \times n$ elements. Each matrix is divided into tiles of $ts \times ts$ elements.

Figure 2.3 shows its source code implementation on a shared-memory CPU system. The *dgemm* function is a task that computes the partial product of each tile by calling the CBLAS

in more detail in Chapter 4.

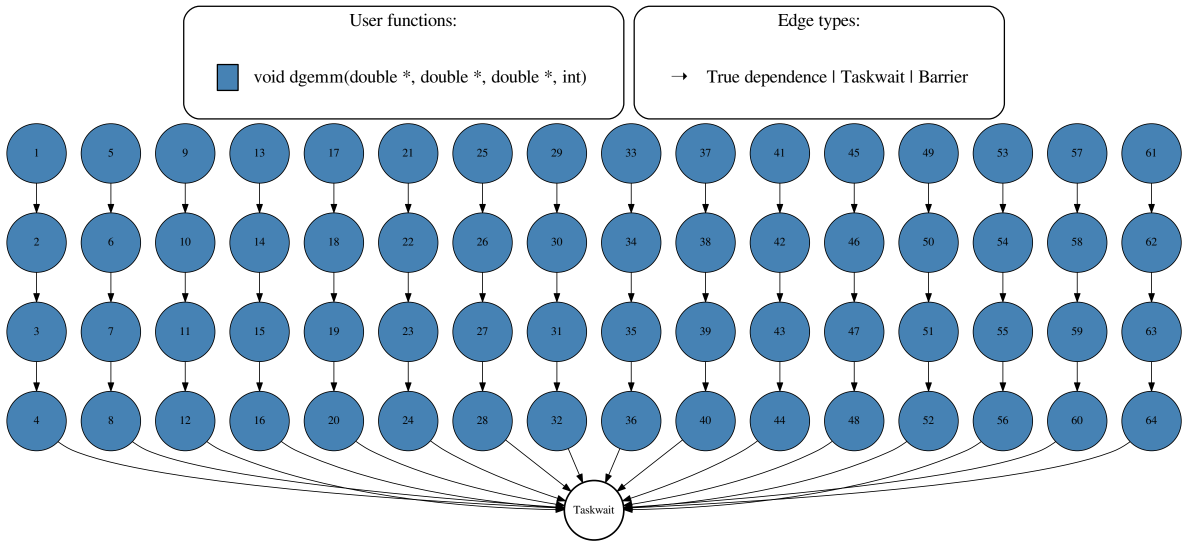

User functions:
void dgemm(double *, double *, double *, int)
Edge types:
True dependence | Taskwait | Barrier
Taskwait

Figure 2.4: Data-dependency graph for a tiled matrix multiply algorithm

library. The **in** and **inout** clauses ensure that the partial computations over the different tiles are done in the correct order. The **taskwait** at the end ensures that all tasks have been run at that point.

Figure 2.4 illustrates the resulting data-dependency graph built at runtime for matrices of 4×4 tiles. Nodes represent tasks and arrows represent data dependencies: the target node of the arrow is a successor of the source node. Nodes have an identification number according to the order of task creation. Each dependency chain corresponds to the computation of one tile of the output C matrix. The last node corresponds to the **taskwait** synchronization.

2.2 Applications

This section describes all the applications that have been used in the context of this thesis. These applications have been carefully chosen depending on their characteristics to evaluate the different contributions presented in this work. They are presented in alphabetical order and only their general description is provided below; the specific data set sizes and/or any configurations are described in the corresponding sections where they have been evaluated.

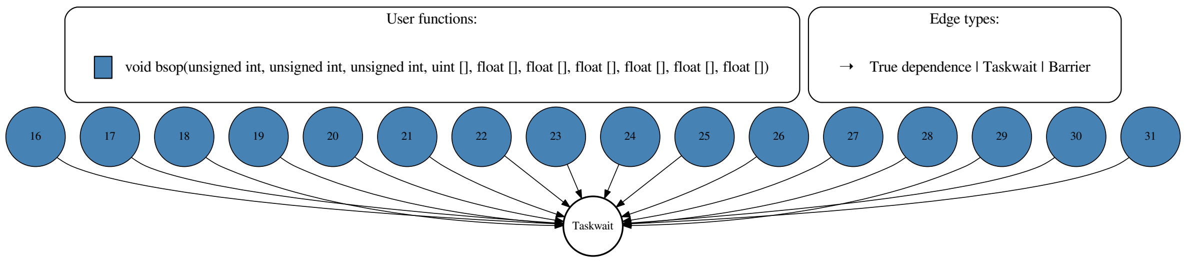

User functions:
void bsop(unsigned int, unsigned int, unsigned int, uint [], float [], float [], float [], float [], float [], float [])
Edge types:
True dependence | Taskwait | Barrier
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
Taskwait

Figure 2.5: Data-dependency graph for a Black-Scholes algorithm

2.2.1 Black-Scholes

The BlackScholes benchmark computes the pricing of European-style options. In a GPGPU context, it is a very data intensive benchmark as the kernel computation is very small with respect to its data input size. Even so, it is worth to be executed on a GPU due to its kernel speed-up with respect to the CPU (∼300x).

In the OmpSs implementation, data is divided into smaller, independent chunks so that tasks operate over these chunks and run in parallel to compute the final output result. Figure 2.5 illustrates the data-dependency graph of the application with data divided into 16 chunks.

2.2.2 Cholesky Factorization

The Cholesky factorization is a matrix operation commonly used to solve normal equations in linear least square problems. it mainly calculates a triangular matrix (L) from a symmetric and positive definite matrix (A). The product of this triangular matrix L and its transposed copy is A: $Cholesky(A) = L$, where $L \cdot L^t = A$.

The source code is the main algorithm of a tiled Cholesky factorization. The matrix A is organized in tiles. The computation is done inside a set of nested loops that operate on these tiles by calling four different kernels: *dpotrf*, *dsyrk*, *dgemm* and *dtrsm*.

In the OmpSs implementation, each kernel is annotated as a task and task data dependencies are managed by the OmpSs runtime.

Figure 2.6 shows the data-dependency graph of this algorithm for a matrix of 8×8 tiles. In order to get good performance in this application, it is important to schedule carefully the execution of *dpotrf* tasks, because in Cholesky's task graph, there are some points where all the following tasks depend on the *dpotrf* task. So, it acts like a bottleneck and if it is not run as soon as its data dependencies are satisfied, there is less parallelism to exploit and, thus, we observe a slowdown in application's performance.

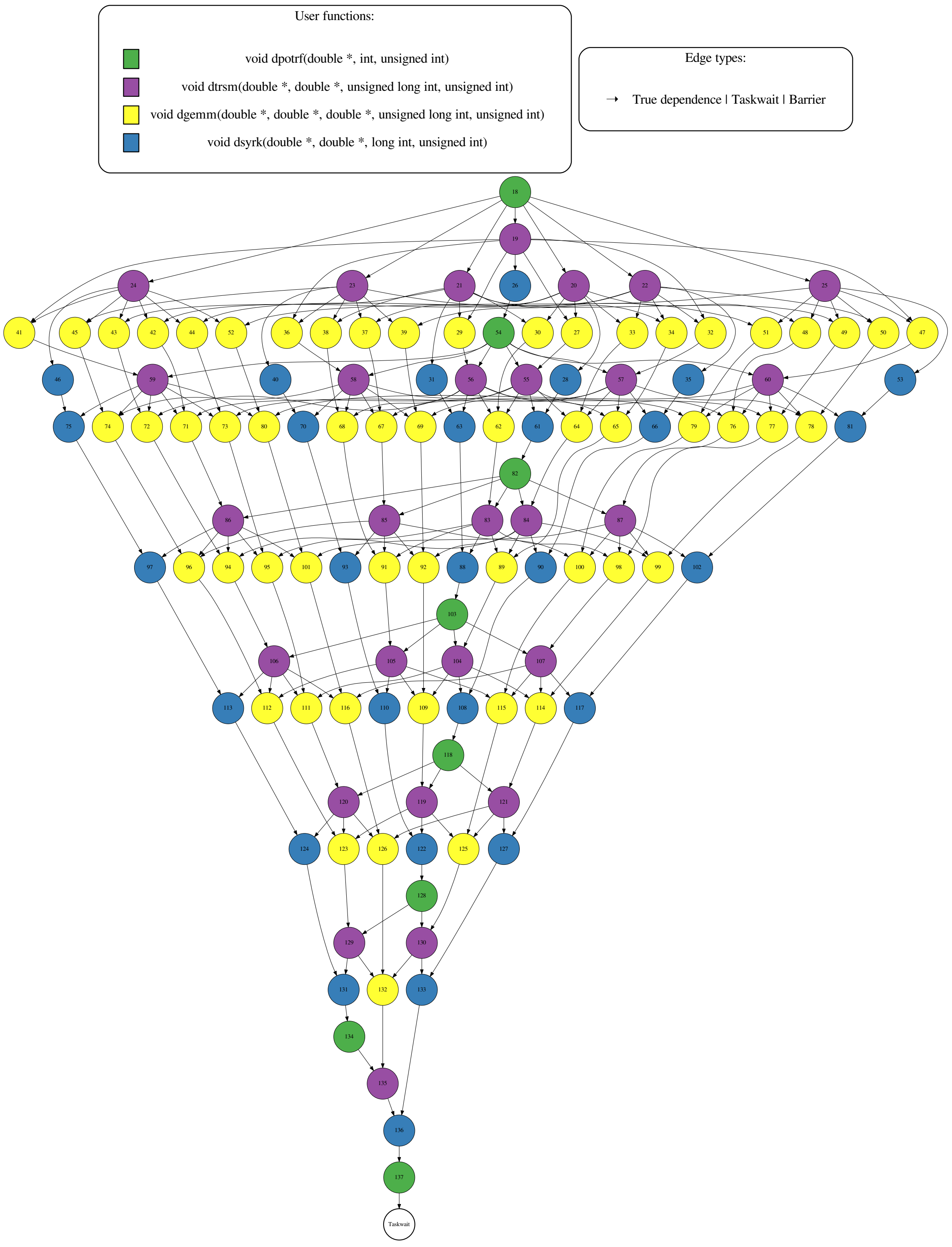

User functions:
🟩 void dpotrf(double *, int, unsigned int)
🟪 void dtrsm(double *, double *, unsigned long int, unsigned int)
🟨 void dgemm(double *, double *, double *, unsigned long int, unsigned int)
🟦 void dsyrk(double *, double *, long int, unsigned int)
Edge types:
→ True dependence | Taskwait | Barrier
Taskwait

Figure 2.6: Data-dependency graph for the Cholesky factorization

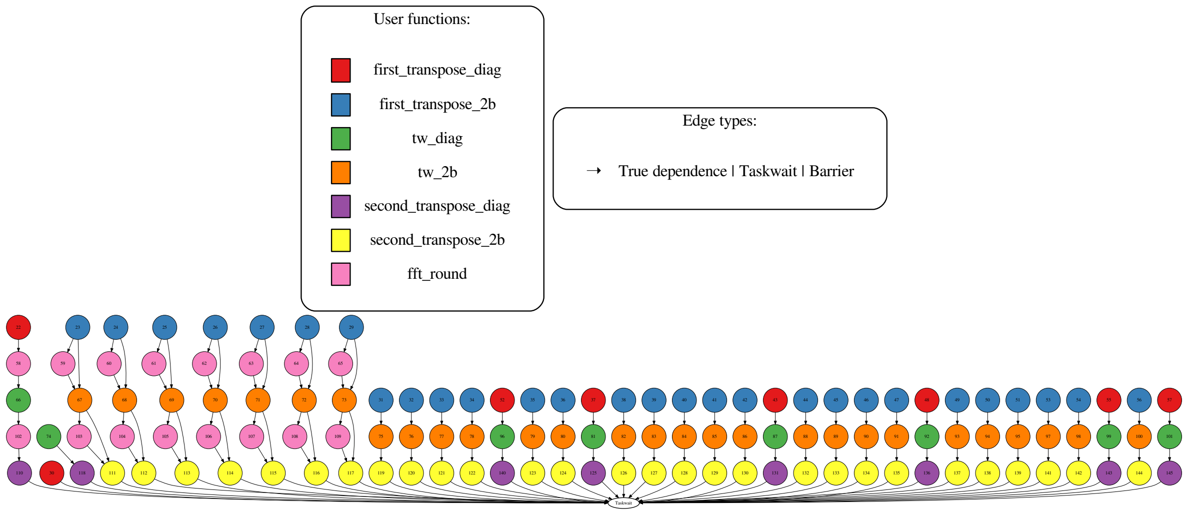


Figure 2.7: Data-dependency graph for the FFT1D transformation

2.2.3 FFT1D

The Fast Fourier Transform 1D (FFT1D) application measures the floating point rate of execution of the double precision complex one-dimensional Discrete Fourier Transform (DFT). The data is distributed in a two-dimensional array of complex double precision elements. The first step of the algorithm performs an in-place transposition of the data, after this, an FFT1D round is applied to each of the rows of the data. The next step is to transpose again the data and to apply a twiddle factor, to follow with a second round of FFT1D on each row. Finally, a last in-place transpose obtains the final result.

In the OmpSs implementation, each of the steps is translated into several tasks that operate on several rows of the matrix. The parallelization of the transpose and the twiddle+transpose are also implemented using tasks that operate on sub-blocks of the matrix. Figure 2.7 represents the data-dependency graph for this algorithm using a matrix of 8×8 tiles.

2.2.4 Krist

This application computes crystallographic normalized structure factors. Data are represented on three arrays: two of them are read and the other one is used to write the results. The size of these arrays is determined by the number of atoms and the number of reflections used in each execution.

In the OmpSs implementation, these arrays are divided into smaller chunks and tasks perform the computation on these chunks, so there is almost no data sharing between them. Figure 2.8 illustrates the data-dependency graph of running Krist for 10 iterations over arrays divided into 8 chunks.

2.2.5 N-Body Simulation

The N-Body simulation is a molecular dynamics computation where a system of bodies (atoms, molecules) is allowed to interact for a period of time. The result of the simula-

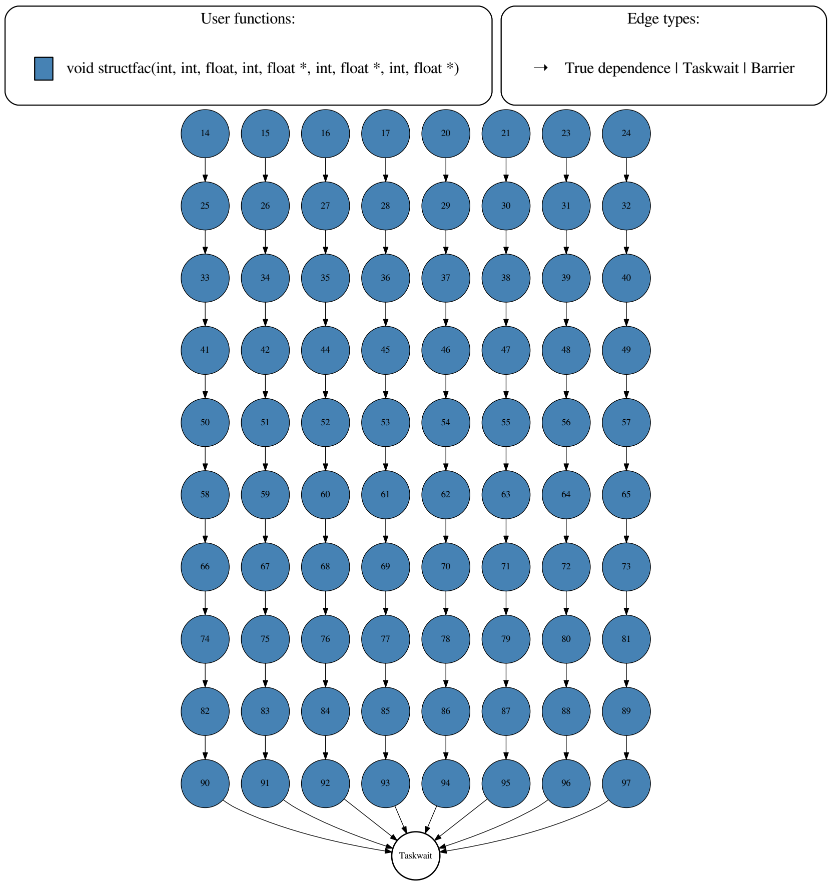


Figure 2.8: Data-dependency graph for Krist application

tion gives a view of the motion of the bodies whose trajectories are determined by forces between bodies and their potential energy.

A CUDA native implementation is distributed with CUDA SDK examples [12, 42]. This code has been transformed into an OmpSs application by adding **`task`** directives around GPU kernel calls with the appropriate data directionality clauses and by removing all explicit data transfers and GPU-management related code.

This simulation is memory bound and its performance is limited by the amount of data that needs to be transferred between GPU devices after each iteration. Figure 2.9 shows the data-dependency graph of 3 iterations of this simulation and partitioning data into 16 chunks.

2.2.6 PBPI

PBPI is a parallel implementation of a Bayesian phylogenetic inference method for DNA sequence data. This solution is based on the construction of phylogenetic trees from DNA or AA sequences using a Markov Chain Monte Carlo (MCMC) sampling method. There are two factors that determine the computation time: the length of the Markov chains used later to approximate the probability of the phylogenetic trees and the time actually needed to evaluate the likelihood values at each generation.

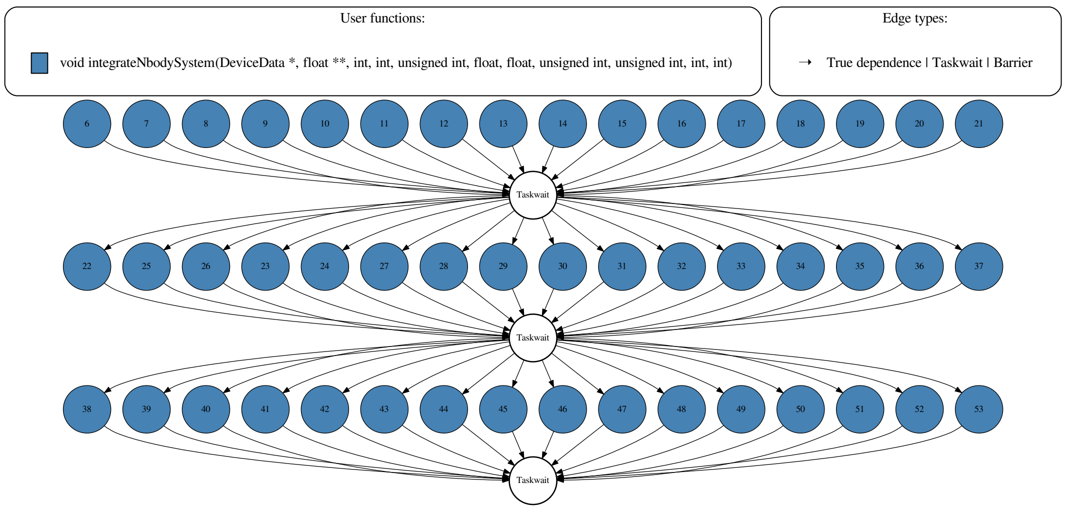

User functions:
void integrateNbodySystem(DeviceData *, float **, int, int, unsigned int, float, float, unsigned int, unsigned int, int, int)
Edge types:
True dependence | Taskwait | Barrier
Taskwait

Figure 2.9: Data-dependency graph for N-Body simulation

In the OmpSs implementation, three different tasks are defined for each of the three computational loops that account for the majority of the execution time of the program. These loops evaluate the likelihood values and one task is created for each iteration of each loop. Figure 2.10 represents a simplified data-dependency graph of this application for the first three iterations of the likelihood loops (thousands of tasks are created in the real application).

2.2.7 Perlin Noise

Perlin noise is a gradient noise used to increase the appearance of realism in computer-generated images. The implementation involves three steps that are computed for each pixel of the image: grid definition, computation of the dot product between the distance-gradient vectors and interpolation between these values.

In the OmpSs implementation, a two-dimensional input image is divided into several stripes containing several rows of pixels and each task computes the noise for one of these stripes. Figure 2.11 illustrates the data-dependency graph of Perlin Noise for an image divided into 8 stripes and applying the algorithm twice (2 iterations).

2.2.8 STREAM Benchmark

STREAM is an HPC Challenge (HPCC) [43] benchmark that measures memory bandwidth for simple kernels, intended for use with large data sets. It performs four simple operations on three one-dimensional arrays: copy, scale, add and tri-add.

In the OmpSs implementation, four different tasks are defined to perform each of the four operations. The arrays are divided into several chunks and each task operates on one of these chunks. Then, tasks that compute the same operation do not share data and can be run in parallel. However, they must synchronize between different operations. Figure 2.12 shows the data-dependency graph of STREAM with arrays divided into 16 chunks and running 2 iterations of the benchmark.

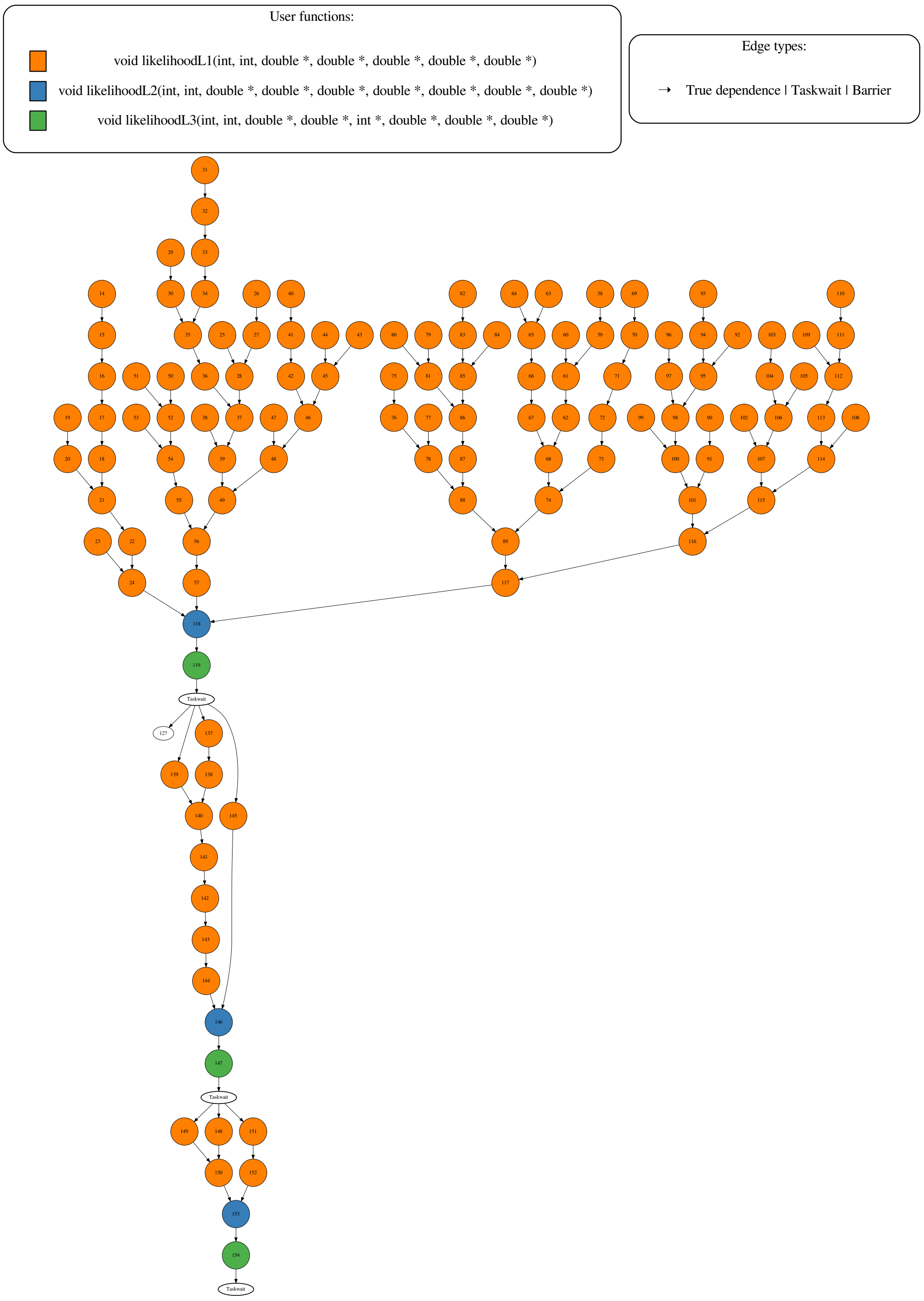

User functions:
void likelihoodL1(int, int, double *, double *, double *, double *, double *)
void likelihoodL2(int, int, double *, double *, double *, double *, double *, double *)
void likelihoodL3(int, int, double *, double *, int *, double *, double *, double *)
Edge types:
True dependence | Taskwait | Barrier
Taskwait

Figure 2.10: Simplified data-dependency graph for PBPI application

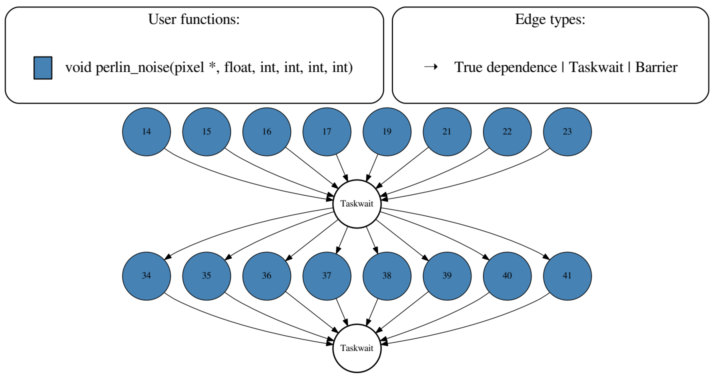

User functions:
void perlin_noise(pixel *, float, int, int, int, int)
Edge types:
True dependence | Taskwait | Barrier
14
15
16
17
19
21
22
23
Taskwait
34
35
36
37
38
39
40
41
Taskwait

Figure 2.11: Data-dependency graph for Perlin Noise application

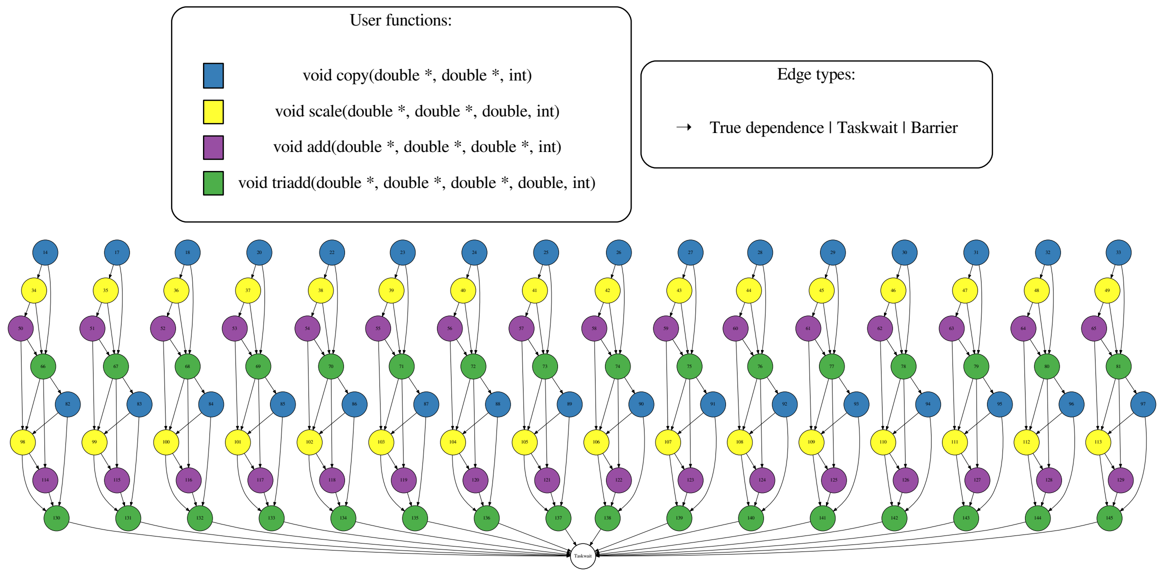

User functions:
void copy(double *, double *, int)
void scale(double *, double *, double, int)
void add(double *, double *, double *, int)
void triadd(double *, double *, double *, double, int)
Edge types:
True dependence | Taskwait | Barrier
Taskwait

Figure 2.12: Data-dependency graph for STREAM benchmark

2.2.9 Tiled Matrix Multiply

This application performs a dense matrix multiplication of two matrices and stores the result into another matrix: $A \times B = C$. For simplicity, all matrices are square matrices and are divided into square tiles as well. Matrix multiply is a well-known algorithm that requires a significant amount of data movements, but also benefits a lot of locality optimizations. This algorithm and its OmpSs implementation have already been explained in Section 2.1.6.

Chapter 3

State of the Art

Heterogeneous architectures that combine different types of computational units (e.g., GPGPUs and traditional CPU processors) are becoming popular mainly due to their high performance capabilities. However, the programmability of these systems is not trivial for the programmer who needs to take into account several aspects such as parallelism, different programming styles or different coexisting memory spaces.

Therefore, many proposals have arisen in the last years to address such complexity. This chapter collects the most relevant ones related to this thesis research work. The contents are divided into different conceptual categories and ordered by relevance. Those proposals that could fit into more than one category are classified according to their main target.

3.1 Task-based Programming Models and Languages

With regard traditional node-level parallel programming models, several approaches have been wide-adopted by programmers to parallelize their applications. As the use of heterogeneous systems is increasing its popularity, these models have revised and extended their specifications in order to support these systems as well.

3.1.1 OpenMP

OpenMP [28] is one of the most popular parallel programming models. Designed for productivity, it defines a standard for a programming model where some parts of the code are executed sequentially, and only those parts of the code (where the programmer has explicitly introduced some directives) are executed in parallel. It supports C, C++ and Fortran and is based on adding some annotations to the source code: compiler directives, which are translated into calls to the OpenMP runtime library routines.

At runtime, there is one thread that executes the sequential regions of the code, and spreads into several threads when a parallel region is found. At the end of the region, all threads join together again and a single thread continues the sequential execution until it reaches another parallel region, or the end of the application.

It also offers the possibility of tuning the scope of parallel-region variables (shared among threads, by default). They are used to control when variables must be shared or thread-private, and how data is transferred and returned from the parallel regions of the code.

OpenMP has been typically used in shared memory systems to execute *do* or *for* loops in parallel. Nevertheless, task-based parallelism support was later included in its version 3.0 [44]. In addition, it has been recently extended in its latest version 4.0 with additional clauses to express data dependencies between tasks. This version includes accelerator support as well: the standard is defined to support a wide variety of compute devices. OpenMP API provides mechanisms to describe regions of code where data and/or computation should be moved to another computing device. Several prototypes with accelerator support have already been implemented. The support for specific accelerator architectures depends on each vendor implementation: for example, the Intel compiler offers support for offloading OpenMP code to Intel Xeon Phi.

Although OpenMP is mainly focused on shared memory architectures, it is not unusual to combine it with other programming models, like MPI, to target distributed memory systems [45].

3.1.2 Cilk Plus

Cilk Plus [46] is a task-based general-purpose programming language for multi-threaded parallel programming. It is based on C/C++ language and supports task nesting and parallel loops as well. There are two main keywords: *spawn*, to identify tasks and *sync*, to wait for *spawned* tasks. There is no data dependence detection mechanism between tasks, but *Cilk hyperobjects* can be used to solve data race problems arisen in global variable accesses. Then, respecting data dependencies is the programmer's responsibility. Cilk scheduler is based on a work-stealing model, capable of exploiting data locality.

Cilk Plus runs on symmetric multiprocessors (SMPs) and Intel Xeon Phi coprocessors. However, the thread pools on the host processor and the Xeon Phi are totally separate and work cannot be stolen between these units. The model targets single-node systems with shared memory, but, like OpenMP, can be combined with other models to execute on distributed memory systems.

3.1.3 Chapel

Chapel [47] is a parallel programming language with the objective of improving the productivity of programmers. It is an imperative block-structured language designed from scratch. Its aim is to make easier the programmability of large-scale computers at the same time that keeps or even improves the performance of current portable programming models.

The proposal is based on a multi-threaded execution model with high-level abstractions for data and task parallelism, concurrency and nested parallelism. It allows reusing code and prototyping through an object-oriented design and features for generic programming.

Chapel has been designed for an ideal system with a global shared address space, hardware support for multi-threading, a high-bandwidth low-latency network and latency-tolerant processors. Thus, when used in a machine that lacks some of these aspects, it gets less performance than expected.

3.1.4 Sequoia

Sequoia [48] is a programming language focused on machines with a hierarchical memory configuration. The aim of Sequoia is to be portable across such kind of machines. It provides a source-to-source compiler and a platform-specific runtime.

The model exposes the memory hierarchy to the language, and provides mechanisms to establish communications between different memory modules. Tasks are self-contained units of computation and are used to express parallelism. They also contain additional information about communication and their specific working sets. There are two types of tasks: *inner*, to describe how to partition data and work into sub-tasks, and *leaf*, to perform computations. The programmer is able to provide several implementations of a task and specify which implementation will be executed according to the context within the task is called.

The framework was first implemented for Cell/B.E. blades and distributed-memory cluster platforms. GPGPU support [49] was later introduced as well. The GPGPU support is implemented on top of CUDA: the CUDA abstract machine model is ignored and instead the Sequoia knowledge of the application is used to target the device directly. Thus, the runtime only launches as many cooperative thread arrays (CTAs) as there are streaming multiprocessors (SMs) on the device. The Sequoia compiler is then able to map tasks directly onto the SMs rather than simply launching CTAs and trusting the hardware to schedule them efficiently.

3.2 Accelerator Programming

The number of proposals for accelerator programming is increasing year after year, as more programmers tend to offload parts of their applications to accelerators, like GPGPUs or the Intel Xeon Phi. Since accessing an accelerator requires specific driver support, most of these proposals have been implemented on top of the software/driver support provided by the accelerator vendor. For example, many approaches targeting GPGPUs have been implemented on top of CUDA.

3.2.1 CUDA

nVIDIA GPUs are one of the most popular accelerators and are extensively integrated in HPC clusters. Compute Unified Device Architecture (CUDA) [12] is almost the de-facto standard for programming GPUs. The programmer has to write specialized pieces of code (called CUDA *kernels*) that are executed concurrently by many threads on the GPU.

With CUDA, the programmer is not only responsible for writing the application code and computational kernels, but also for performing memory allocation and managing data transfers between host memory and device memory to achieve optimal performance.

Since the first CUDA 1.0 release in 2007, the framework has evolved in many different aspects to fulfil and attract new users. For example, *Unified Memory Access* (UMA) was introduced in CUDA 6.0: a pool of *managed memory* is created and accessible from both host and device sides. The programmer always uses the same pointer address to access memory on both sides and the CUDA runtime performs internally the necessary memory transfers between the different memory spaces. Then, there is no need to issue any explicit memory transfer. UMA has dramatically improved the easiness of CUDA programming, but still its performance is questioned among the expert programmer community [50].

CUDA is usually combined with MPI as well to distribute computations among GPGPU clusters [51, 52, 53, 54, 55].

3.2.2 OpenCL

Open Computing Language (OpenCL) [13] has been proposed as an alternative to CUDA to program accelerators. They both offer very similar functionalities, but OpenCL's strongest aspect is portability, as, in addition to GPGPUs, it supports programming general purpose multicores, Intel Xeon Phi cards, FPGAs and Digital Signal Processors (DSPs) as well. However, it offers a very low-level API to the programmer, exposing them to explicitly manage data and thread execution. For example, it is programmer's responsibility to build the program executable or to move data between cores and accelerators. In addition, although application source codes are portable across different platforms, performance portability is not guaranteed and, in fact, experiments demonstrate slow-downs for several applications [56, 57].

3.2.3 HSA

The Heterogeneous System Architecture (HSA) Foundation defines a standard to unify the architecture of accelerators [58]. The HSA specification establishes a set of requirements regarding several architecture aspects, like virtual memory, memory coherency, offloading mechanisms and power-efficient signals. HSA defines different components [59] (i) the HSA system architecture, which specifies a common base to build portable applications targeting accelerators, (ii) a high-level compiler that generates HSA Intermediate Language (HSAIL) code, (iii) a low-level compiler, called finalizer, that translates HSAIL code into the target architecture code and (iv) the HSA runtime API that provides mechanisms to handle accelerators from the host, such as memory management, computation offload, synchronization, signals, error handling, etc.

The HSA runtime [60] has been designed to offer a common accelerator interface for other programming languages, like OpenCL, OpenMP, Java or a domain-specific language (DSP). These languages offer a way to express parallel regions of code that can be offloaded to an accelerator. Then, the language compiler is responsible for generating the HSAIL code for such parallel regions, including the necessary calls to the HSA runtime to set up and offload the computation. And the finalizer, which must be provided by the accelerator vendor, translates the HSAIL code into accelerator's hardware code.

3.2.4 OpenACC

Open Accelerators (OpenACC) [61] aims at simplifying offloading of tasks to accelerators. It defines a set of functions and compiler directives, similar to OpenMP, to specify parts of a program whose computation may be offloaded to an accelerator, to transfer data between main memory and the accelerator and to synchronize with the execution of those accelerator computations. OpenACC has support for accelerators like Accelerated Processing Units (APUs), GPUs or many-core processors. However, the current OpenACC 2.0 version specification only addresses machines with one accelerator.

The programming model defined by OpenACC allows programmers to create high-level hybrid host+accelerator programs without the need to explicitly initialize the accelerator, manage data or program transfers between the host and accelerator, or initiate accelerator startup and shutdown.

3.2.5 Intel Offload

The Intel C++ compiler [62] supports a set of directives to offload portions of code to an Intel Many Integrated Core (MIC) Architecture. Similarly to OpenMP, it defines several clauses to specify the offload target device and the data needed by the offloaded code, as it will be copied to MIC's memory space. Code offloading can be synchronous or asynchronous and synchronization mechanisms are also offered.

3.2.6 StarPU

StarPU [63] is based on a tasking API and also on the integration of a data-management facility with a task execution engine. With regard to data management, StarPU proposes a high level library that automates data transfers throughout heterogeneous machines [64].

The model defines *codelets* as an abstraction of a task (e.g., a matrix multiplication) that can be executed on a core or offloaded onto an accelerator using an asynchronous continuation passing paradigm. Multicore processors, nVIDIA GPUs, OpenCL devices and Cell/B.E. processors [65] are supported nowadays [66, 67]. Future support for Intel Xeon Phi is advertised on their website [68].

StarPU offers low level scheduling mechanisms (like work stealing) to be used in a high level fashion, regardless of the underlying (possibly heterogeneous) target architecture. The authors are not proposing a programming model, but only a runtime library. Therefore, the programmer is exposed with low-level APIs and execution details.

3.2.7 Offload

Offload [69] is a programming model for offloading portions of C++ applications to run on accelerators. Code to be offloaded is wrapped in an *offload* block, indicating that this code should be compiled for an accelerator, and executed asynchronously as a separate thread.

Call graphs rooted at an offload block are automatically identified and compiled for the accelerator. Data movement between host and accelerator memories is also handled automatically.

3.3 Heterogeneous Schedulers

There have been many proposals to partition workload between the different units of a heterogeneous system. However, only a few of them are able to make such partition dynamically, at runtime, without requiring a previous application profiling. And to the best of our knowledge, there is no dynamic partition proposal that considers more than two different processing units within the same system, like SSMART[1] scheduler does.

[1] SSMART scheduler is one of this thesis contributions and its description can be found in Section 5.4.

3.3.1 MDR

Model Driven Runtime (MDR) [70] is a runtime framework to schedule workloads represented as Direct Acyclic Graphs (DAGs) in heterogeneous parallel platforms. The proposal is based on four criteria, called *SLAC* (Suitability, Locality, Availability and Criticality):

- *Suitability*: which processing unit is faster to execute a given task.
- *Locality*: whether the data needed by a task is present in the memory space of the processing unit.
- *Availability*: when a processing unit will be available to run a task.
- *Criticality*: how the execution of a given task can affect the overall execution of the application.

The runtime combines all the aforementioned criteria with performance models to make decisions such as mapping a task to a processing unit or copying data between memory spaces. The authors show that the four *SLAC* criteria must be considered by a heterogeneous runtime framework in order to achieve good performance under varying application and platform characteristics. MDR has been implemented on top of Intel Thread Building Blocks (TBB) and nVIDIA CUDA.

3.3.2 Qilin

Qilin [71] aims at distributing kernel computations between CPUs and GPUs. An API to write *parallelizable* operations is offered at application level. The Qilin compiler dynamically translates the API calls into native machine codes. It also decides the near-optimal mapping from computations to processing elements using an adaptive algorithm. In order to reduce the compilation overhead, translated codes are stored in a code cache to be reused without recompilation. Once native machine codes are available, they are scheduled to run on the CPU and/or GPU by the Qilin scheduler.

The runtime component creates a directed acyclic dependency graph of kernels as the application is being run. The runtime determines which kernels can be run in parallel and maps them dynamically to available processing units (either CPU or GPU). Qilin uses an analytical performance model to determine the execution time of individual kernels on specific accelerators, but it can only exploit parallelism within a single basic block.

3.4 Tools for Heterogeneous Platforms

Writing and offloading code to an accelerator is not a trivial process, since it includes several tasks, like translating sequential code into massively parallel code or dealing with a hierarchical memory distribution. Therefore, many tools have arisen to simplify these tasks.

3.4.1 hiCUDA

hiCUDA [72] is a high-level directive-based language for CUDA programming. Programmers can use hiCUDA directives in a sequential source code to give hints to the compiler about regions of code that can be exploited on GPUs and about data motion. Then, the authors also present a source-to-source compiler that transforms a hiCUDA program into a CUDA program. The resulting source code can be compiled with the CUDA compiler.

3.4.2 CAPS HMPP

The CAPS HMPP [73] toolkit is a set of compiler directives, tools and software runtime that supports parallel programming in C and Fortran. HMPP works based on *codelets* that define functions that will be run in a hardware accelerator. These codelets can either be hand-written for a specific architecture or be generated by some code generator. Both GPGPU and Xeon Phi devices are supported. This work was later derived to the OpenHMPP standard [74] that defines a set of compiler directives for heterogeneous computing.

3.4.3 ispc

The Intel SPMD Program Compiler (ispc) [75] is a compiler that delivers high performance on CPUs thanks to effective use of both multiple processor cores and SIMD vector units. ispc relies on the idea of GPU programming languages, which have shown that for many applications the easiest way to program SIMD units is to use a single-program, multiple-data (SPMD) model. The compiler supports C and C++ languages and targets both CPU and Intel Xeon Phi architectures.

3.4.4 OpenMPC

Lee et al. propose OpenMPC [76]: an OpenMP-to-CUDA translation system that performs a source-to-source conversion of a standard OpenMP program to a CUDA program and applies various optimizations to achieve high performance.

The compiler interprets OpenMP semantics under the CUDA programming model and identifies kernel regions (code sections to be executed on a GPU). Then, those eligible kernel regions are transformed into CUDA kernel functions. It also inserts the necessary memory transfer code to move data between CPU and GPU.

3.4.5 CUDA-lite

CUDA-lite [77] is a set of tools and source code annotations to better map algorithms to the GPU memory hierarchy. Programmers provide straight-forward implementations of the application kernels using only global memory with some annotations. CUDA-lite tools do the appropriate transformations automatically to exploit local memory hierarchy and minimize memory latency through coalesced accesses.

3.4.6 UPC for GPU Clusters

With the objective of tackling clusters of GPUs, Chen et al. present an extension to Unified Parallel C (UPC) [78] with hierarchical data distribution [79]. The approach extends the semantics of *upc_forall* to support multi-level work distribution. This work also presents features based on compiler analysis such as affinity-aware loop tiling and the runtime implementation of a unified data management on each UPC thread to optimize data transfers between CPU and GPU.

Chapter 4
Accelerator Support for OmpSs

One of the contributions of this thesis, as introduced in Chapter 1, is to add support for accelerators in the context of the OmpSs programming model. This support was developed in two steps that were adapted to their contemporary technologies. Therefore, a semi-asynchronous accelerator support was developed as a first step and transformed into a fully-asynchronous approach as the second step.

The reason why a first semi-asynchronous approach was developed is that the existing technology by the time it was implemented did not offer any support to develop a fully-asynchronous approach. In addition, the accelerator support in this first step only included GPGPUs because Intel Xeon Phi had not appeared in the market. The semi-asynchronous approach is described in Section 4.1.

As the existing technologies evolved, it became clear the need to offer fully-asynchronous mechanisms to establish communications between host and accelerators. Therefore, mechanisms like events and callbacks were introduced in CUDA and OpenCL programming languages. Once these new features became available to accelerator programmers, the second fully-asynchronous approach was developed. The validity and efficiency of this second step has been proved with both GPGPU and Intel Xeon Phi accelerators. The details of this second design are explained in Section 4.2.

One of the main goals of the OmpSs accelerator support is the ease of managing accelerators from the programmer point of view. On the one hand, all memory transfers needed by offloaded tasks from/to the device memory space are managed by the runtime, in such a way that the programmer does not have to worry about them. On the other hand, the runtime is able to transparently manage several devices (possibly of different types) at the same time. Even though developing a multi-accelerator application is more complicated because the device management must be done by hand, creating several host threads, synchronizing them explicitly, etc., the OmpSs accelerator support allows running the same application equally on a single device or several devices without modifying its source code. This means that switching from a single-accelerator application to a multi-accelerator version is effortless for the programmer.

In addition, the OmpSs accelerator support offers several configurable run time parameters. The specific options depend on the accelerator architecture, such as the ability to set the number and accelerator types that Nanos++ runtime can use, limit the amount of memory available on each accelerator, etc. In general, it is better to have as much device memory available as possible to maximize performance, but programmers may want to use extra device memory on their own, or even limit the amount of memory used in order to reduce energy consumption.

4.1 Semi-asynchronous Accelerator Support

This section explains in detail the first prototype designed as an OmpSs extension to support GPGPU architectures. First, the general design that could target different types of accelerators is presented. Then, the specific GPU support characteristics and features are explained. Most part of the work was developed inside the Nanos++ runtime library, but some support from Mercurium compiler was also needed. Although the compiler support needed by both the semi-asynchronous and the fully-asynchronous approaches is exactly the same, it is described in this section for completeness.

4.1.1 Accelerator Agnostic Design

The Nanos++ independent layers explained in Chapter 2 interact with the architecture-dependent components to execute tasks on the processing units of the system. The accelerator support component was designed in the context of this thesis to provide the base model for the architecture-dependent components that target accelerators. Its accelerator agnostic design takes into account the general properties that characterize accelerators, making it valid for different kinds of device architectures.

The architecture-dependent component for each accelerator is responsible for the following actions:

- **Device data movements**: Issue data transfers between host memory and its device memory space.
- **Execute tasks**: Usually, the task targeting the device will be launched from the host side and will run asynchronously on the device.
- **Synchronize with device operations**: This basically includes synchronizations with kernel launches and memory transfers, that may or may not be asynchronous.

4.1.1.1 Overlap of Data Transfers and Computations

In general, accelerators offer the possibility to hide the overhead of data movements between memory spaces by overlapping device computations with data transfers. Hence, our accelerator agnostic design is based on this feature. It is important to notice that device operations must be asynchronous from the host point of view in order to be able to overlap them.

The OmpSs software cache is prepared to support asynchronous operations, working as a *non-blocking cache*, provided that the underlying architecture-dependent layers support it. Then, the cache will not wait for data transfers to complete and will return control to the runtime so that it can do other operations, like launching a kernel on the device. When the software cache works in the non-blocking mode, it expects a later transfer completion notification from the architecture-dependent component. From the point of view of Nanos++ independent layers, asynchronous data transfers are not completed until the software cache is notified. This implies that, for example, the task execution cannot be safely triggered until the software cache has verified that all task data is present and updated on the device. Thus, it is important for the architecture-dependent component to notify transfer completions as soon as possible.

4.1.1.2 Data Prefetch

Prefetching and overlapping are two actions that are usually combined because data prefetching is just a way to make overlapping of data transfers with computations possible. In the case of OmpSs applications, tasks are annotated with data-directionality clauses, therefore the runtime is aware of what pieces of data each task needs at its creation time. This information can be used by the architecture-dependent component to issue task data transfers in advance. Hence, task data will be available on the device by the time the task is offloaded, so there will be no delays.

4.1.1.3 Task Execution Flow

We define a task execution cycle as the set of needed actions to run a task. Then, the execution cycle of a task can be divided into the five stages of the task life cycle described in Section 2.1.4.1: instantiation, ready, active, run and completion.

The SMP-dependent component, used to run tasks on CPUs, executes one task cycle after another as long as there are tasks pending to be run on a CPU. However, if the same approach is followed for accelerators, there will be no overlapping at all. Then, the accelerator agnostic design interleaves several task cycles to overlap data transfers with computations on a device in the following way: for every task code offloaded at the run stage, if possible, another two stages from different task cycles happen:

- **Completion stage for the previous task**: if needed, data transfers from the device to the host will happen in this stage, so that they can potentially overlap with the current task execution.
- **Active stage for the next task**: data transfers from host memory to device memory are issued in this stage, if needed. Then, there is another opportunity to overlap data transfers with the execution of the current task.

For this semi-asynchronous approach, we assume that the only way to synchronize the host and the device is by means of blocking the host side and waiting for the device to finish its operations. Thus, on one side, issuing many asynchronous device operations is good for overlapping, but on the other side, host-device synchronizations are needed to notify the runtime independent layers when device operations finish.

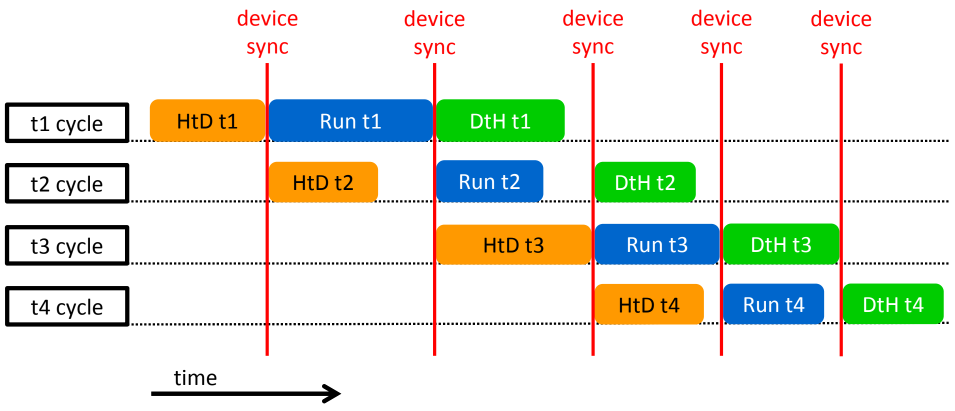


Figure 4.1: Semi-asynchronous task execution flow

Therefore, we found a balance between synchronization and overlapping for the accelerator-agnostic design: a single synchronization point is introduced for every task cycle, at the end of the run stage of the current task. This way, three simultaneous operations are allowed (two data transfers and one kernel execution) and synchronized at the same time. Figure 4.1 illustrates the execution flow of four tasks: *t1*, *t2*, *t3* and *t4*. Only the representative stages involved in the overlapping process are shown as colored boxes:

❑ Orange boxes: Labelled as *HtD* (host-to-device), represent the active stage, where data may be transferred to the device memory.

❑ Blue boxes: Labelled as *Run*, represent the run stage, where the task is executed on the device.

❑ Green boxes: Labelled as *DtH* (device-to-host), represent the completion stage, where data may be transferred back to the host memory space.

❑ Vertical red lines: Indicate the synchronization points after each run stage.

As the runtime has more tasks to run, the number of operations that can be overlapped increases. For example, the optimal overlapping is achieved in the run stages of tasks *t2* and *t3*. Since a single synchronization point is introduced for potentially three different operations, the synchronization completes when the three operations are finished. Then, if one of the operations takes considerably much longer than the other two, it will negatively affect performance. But, unfortunately, it is impossible to detect this situation *a priori*.

4.1.2 OmpSs GPU Support

This section explains the specific implementation of the OmpSs GPU semi-asynchronous support. It is based on the accelerator agnostic design and specialized for GPGPU devices.

4.1.2.1 GPU Accelerator Initialization

The Nanos++ GPU dependent layer works on top of the nVIDIA CUDA runtime library. When an OmpSs application is executed, the Nanos++ library is loaded and initialized right before the *main* program function is called. The GPU dependent layer detects the number of CUDA-capable GPGPUs during the Nanos++ initialization and creates a GPU helper thread for each available GPGPU in the system. Helper threads run on the host and are associated to one GPGPU accelerator. These threads will be used through the entire application execution to manage, issue and synchronize all the operations related to its device.

At this point, the GPGPU devices are initialized and their characteristics are captured to be used later by the runtime. For example, the Nanos++ software cache needs to know the exact amount of GPU device memory to detect when the memory is full and whether it is possible to transfer more data or not. CUDA streams are created during initialization as well.

4.1.2.2 GPU Memory Management

Allocations and deallocations of GPU global memory from the host are costly and take a considerable amount of time. In order to avoid such overheads that can penalize application's performance, the whole GPU memory is allocated during the Nanos++ initialization as one piece and is then internally managed by the runtime.

Data coherence is managed by the runtime software cache, explained in Section 2.1.4.4. Whenever the software cache detects that a data transfer involving a GPU device is needed, a data transfer request is sent to its helper thread. The helper thread keeps a list of the requested memory transfers and such transfers are issued, if possible, during the run stage of a task, so that they can be overlapped with the task kernel execution. If the device is idle and has no tasks to run, the transfers are issued as soon as they are received to avoid delays and unnecessary waits from the other runtime components.

4.1.2.3 Overlap of Data Transfers and GPU Computations

In CUDA, data transfers and kernel launches must accomplish several restrictions so that the hardware can overlap transfers with computations. The complete list of restrictions can be found in the CUDA C Programming Guide [12], but, in short, two operations can overlap if they meet the following requirements:

- ❏ The GPGPU hardware must have support for: overlap of data transfer and kernel execution, concurrent kernel execution and/or concurrent data transfers.
- ❏ Operations must be asynchronous.
- ❏ Operations must be issued to a stream different than the NULL stream.
- ❏ All host memory involved in copy operations must be page-locked.

The contemporary GPGPU devices to this design supported, at most, overlapping the following three operations: one host-to-device data transfer, one device-to-host data transfer and one kernel execution. But, for example, two kernels could not run simultaneously on the GPGPU. Then, the GPU helper thread uses three different CUDA streams to issue operations: two streams for host-device data transfers (one for each transfer direction) and one stream for kernel launches. Alternatively, since operations issued to the same stream are executed in

```
// Allocate page-locked memory
void * nanos_malloc_pinned_cuda (size_t size);

// Free page-locked memory
void nanos_free_pinned_cuda (void *address);
```

Figure 4.2: Nanos++ API functions for allocating and deallocating page-locked memory

a FIFO order, a different approach can be used: data transfers needed by a kernel are issued in the same stream, right before the kernel launch. Then, output data transfers are issued in the same stream as well, right after the kernel launch. While this alternative approach is also valid, the chances to overlap operations are lower due to hardware limitations [80].

Since the Nanos++ runtime cannot check whether a user memory region is page-locked or not, an auxiliary page-locked memory buffer is allocated and used as an intermediate buffer to perform memory copies between host and device: for host-to-device transfers, first, the original user memory is copied to the intermediate buffer (using a regular memory copy) and then, an asynchronous copy is issued from the intermediate buffer to the device memory. For device-to-host transfers, the same steps are followed in the reverse order. The auxiliary page-locked memory buffer is allocated at Nanos++ initialization and its contents are invalidated after each memory copy.

For performance reasons, the GPU dependent component offers two Nanos++ API functions to allocate and deallocate page-locked host memory through the Nanos++ runtime. The runtime keeps a list of the page-locked regions that is updated every time these functions are invoked from the user application. Then, asynchronous memory copies involving memory regions that are present in the page-locked region list can be issued directly, without the extra copy to the intermediate memory buffer. These copies are obviously faster and can increase application performance. The syntax of these functions is shown in Figure 4.2 and for compatibility, it is equivalent to the C *malloc()* and *free()* functions. The runtime calls internally the CUDA library to allocate and deallocate page-locked memory.

4.1.2.4 Data Prefetch and Task Execution Flow

In order to follow the accelerator agnostic model, once the GPU dependent component has offloaded a task T to its GPGPU device, it asks the runtime scheduler for a new task T_{next}. Then, if a new task is assigned to the accelerator, T_{next} active stage begins and its input data transfers are issued.

Moreover, if a previous task T_{prev} was executed before T, the completion stage for T_{prev} begins at this point. Hence, the GPU dependent component is able to overlap two data transfers (one in each direction) with device computation. According to contemporary GPGPU features, this is the maximum number of operations that can be overlapped.

Once the three operations are issued, the GPU dependent component waits until they have finished by means of CUDA stream synchronization mechanisms.

```
// Get the appropriate CUDA execution stream
cudaStream_t nanos_get_kernel_execution_stream ( void );

// Get the CUBLAS context associated to the current GPU device
cublasHandle_t nanos_get_cublas_handle ( void );
```

Figure 4.3: Nanos++ API additional functions to manage GPU computation offload

4.1.2.5 GPU Computation Offload

Since all the CUDA stream and synchronization management is done inside the runtime and completely hidden to the programmer, the GPU computation offloading must be issued to the appropriate CUDA stream that the runtime expects. Thus, the GPU dependent component offers an additional Nanos++ API function to query for the appropriate CUDA stream where the computations must be offloaded. The syntax of this call is shown in Figure 4.3. As explained in the next section, the Mercurium compiler is able to automate this step and hide it from the programmer point of view.

Nanos++ with the GPU support is, in general, compatible with other libraries that implement GPU computations, like CUBLAS or MAGMA. However, the CUBLAS library specification changed from its version 3.2 to 4.0: all API functions now receive an extra parameter that corresponds to a CUBLAS context of type *cublasHandle_t*. The context is an opaque structure internally used by the CUBLAS library that must be initialized at the beginning of the application and destroyed at the end to release its associated resources. It can also be used to associate a specific GPGPU device to the context, so that all the CUBLAS calls receiving that context will be issued to its associated device. The introduction of the CUBLAS context concept conflicts with the OmpSs philosophy, as it forces the user to be aware of which GPGPU has to run each computation.

Consequently, additional support to use the CUBLAS library was required from the Nanos++ library: a CUBLAS context is created for each GPGPU device at Nanos++ initialization and it can be queried by the user through a Nanos++ API function. The syntax of this API function is shown in Figure 4.3. Although we put all our efforts to hide such kind of details from the programmer point of view, programmers need to get the CUBLAS context through the Nanos++ API call each time they call a CUBLAS library function and this must be done inside the task context in order to work properly.

In addition, the user is also required to set the appropriate CUDA stream for each CUBLAS call. Even though we tried to hide this requirement from the programmer, our attempts failed: trying to set the appropriate CUDA stream internally in the Nanos++ runtime had no effect as CUBLAS kernels were not sent to such stream. The only possible way for CUBLAS to send the computations to the appropriate CUDA stream is to set such stream from the user code and inside the task context. Figure 4.4 shows how a CUBLAS library call must be invoked from an OmpSs task: inside the task context, both the CUBLAS context and the CUDA stream must be queried to Nanos++ runtime and then used to set the appropriate computation stream for the CUBLAS library through the *cublasSetStream()* function. Then, the desired CUBLAS function can be called by passing as the first argument the CUBLAS context returned by the Nanos++ runtime API call.

```
#pragma omp target device (cuda) copy_deps
#pragma omp task inout ([n] X)
void scal_task (double * X, int n)
{
    double alpha = 2.0;
    cublasHandle_t handle = nanos_get_cublas_handle();
    cudaStream_t stream = nanos_get_kernel_execution_stream();
    cublasSetStream(handle, stream);
    cublasDscal(handle, n, alpha, X, 1);
}
```

Figure 4.4: Task source code example calling a CUBLAS library function

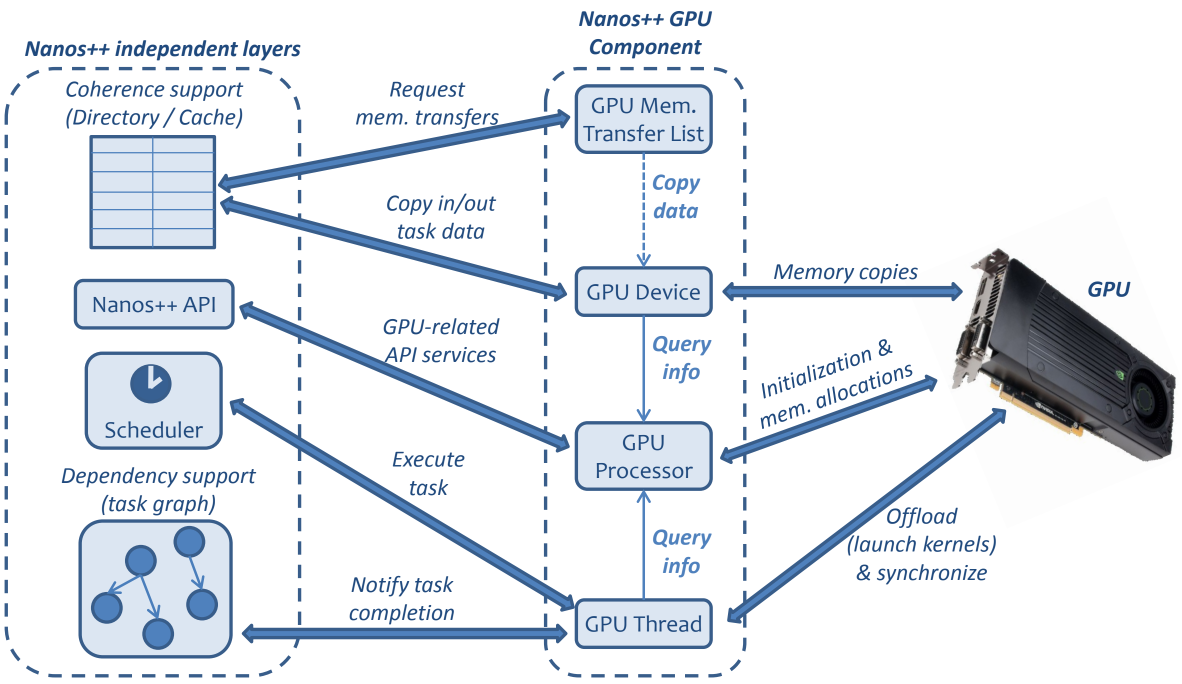


Figure 4.5: Interaction between Nanos++ independent layers and GPU dependent component

Figure 4.5 summarizes the GPU component functionalities and its interaction with the other Nanos++ independent layers. The *GPU Mem. Transfer List* component is used by the coherence layer to request memory transfers of data regions that are needed by other processing units of a different memory space. This situation usually happens when the GPGPU device was the latest producer of a data region and another processing unit (that cannot access the GPU memory space) is going to run a task that needs such data region. This component manages the data requests and interacts with the *GPU Device* component, which is the one that will issue the real memory transfers. The coherence layer also uses the *GPU Device* directly to request the data transfers that are needed by tasks that will run on the GPGPU. The *GPU Processor* is the entity that represents the GPGPU device: it stores the hardware characteristics, like the amount of global device memory, takes care of GPU initialization, like the creation of CUDA streams and manages memory allocations on both the GPU global memory and on the host side as page-locked memory. The other elements of the GPU component also interact with the *GPU Processor* to get the desired information, for example, the *GPU Device* gets the CUDA stream from the *GPU Processor* when it has to issue a mem-

```
// Original kernel call
myKernel <<< grid , block >>> ( param1 , param2 );

// Modified kernel call by Mercurium's CUDA device profile
myKernel <<< grid , block , 0 , nanos_get_kernel_execution_stream() >>> ( param1 , param2 );
```

Figure 4.6: Comparison between original and modified kernel calls by Mercurium

ory transfer. The GPU-related Nanos++ API services also interact with the *GPU Processor* to get the information related to CUDA streams and CUBLAS context and to allocate and deallocate page-locked memory. Finally, the *GPU Thread* is the responsible for launching the task kernels on the GPGPU. It uses the Nanos++ scheduler to get GPU tasks to execute on the device and, once they are completed, notifies the dependency layer. It also interacts with the *GPU Processor* to get the appropriate execution stream for kernels.

4.1.3 CUDA Device Profile for Mercurium Compiler

The programmer needs a small support from Mercurium compiler in order to compile their applications using GPUs. This section explains how Mercurium compiler has been extended to fit these user needs with the CUDA device profile, which is also included in the context of this thesis.

The CUDA device profile has been developed as a component of the Mercurium compiler and participates in the compilation process of an OmpSs application. It is invoked each time the compiler parses the user code and finds a GPU task (annotated with the `device (cuda)` clause). First, it generates a specialized *outline* for the GPU device: a function wrapper[1] that calls the user task. Then, the whole implementation of the GPU task is removed from the original source code file and it is copied to a separate CUDA file (a file with *.cu* extension). Additional symbols and include files are checked and also brought into this new CUDA file if the compiler determines that the GPU task needs them. Finally, this new file is reintroduced in the compiler pipeline with the specialized GPU compilation profile and it is compiled using the nVIDIA *nvcc* native compiler.

The programmer can decide which CUDA stream uses to launch their kernels. However, it is recommended for an OmpSs application to let the Nanos++ runtime make this decision. Otherwise, the host-device synchronization inside the Nanos++ runtime will not work properly. In CUDA, each kernel call must be *configured* through the CUDA extended syntax by placing the kernel configuration parameters between the kernel function name and the kernel arguments, surrounded by `<<<` and `>>>` symbols. CUDA defines two mandatory parameters for the kernel configuration, which are, respectively, the grid and block sizes of the GPU threads running the kernel. Optionally, two more parameters can be specified, which are the amount of shared memory needed by the kernel and the CUDA stream where the kernel must be launched respectively.

With the objective of making the programmer task easier, Mercurium's CUDA device profile is able to recognize CUDA kernel calls and add the necessary code to make the Nanos++

[1] When a task is called, the information about the task environment, like task parameters, are captured in several data structures that are passed to the runtime. Then, when the task is run, the task function wrapper is used, for example, to extract the information of these data structures and call the user task code.

runtime decide the stream to launch the kernel. Figure 4.6 shows how the compiler can transform the kernel call in line 2 into the call in line 5. By adding the Nanos++ API call *nanos_get_kernel_execution_stream()*, the runtime will decide the stream to launch the kernel at run time. However, the CUDA device profile component checks if the original kernel call is already setting a specific stream in order to respect the user's decision: if the user decided to launch the kernel in a specific stream, the compiler does not change this configuration, since it could have collateral effects and lead to incorrect application execution. Then, it is also the user's responsibility to properly synchronize the host with this kernel launch.

It is important to notice that Mercurium does not generate CUDA kernels automatically since CUDA code generation falls out of the compiler scope. However, external tools can be used to generate such kernels and then compile the resulting files with Mercurium.

4.1.4 OmpSs Example Using GPUs

Figure 4.7 shows an OmpSs implementation of a tiled matrix multiply algorithm that offloads the computation to the GPGPU. The equivalent CPU version of this code has been presented in Chapter 2. By just adding a few clauses and providing the GPU kernel implementation, the same code can then target GPGPU devices as well. The **`device(cuda)`** clause specifies that the task should be run on a GPU. The **`copy_deps`** clause indicates that the data specified by the dependence clauses must be available in the GPU memory before the task starts its execution. However, this last clause does not necessarily imply copying all the input and output data of each task right before and after its execution. This allows the runtime to take advantage of different caching and prefetching techniques without the user needing to modify their code.

In this case, the CUBLAS library is called to perform the computation. As explained in Section 4.1.2.5, the Nanos++ API functions are used to set the proper CUDA stream for CUBLAS. Despite the fact that the user is responsible to either write their own kernel code or use an appropriate library, OmpSs takes care of all data movements and kernel synchronizations. Moreover, because these operations are not reflected in the user source code, the same application can be run in a multi-GPU system and the Nanos++ runtime can perform different kinds of optimizations without these being noticed in the source program.

4.1.5 Evaluation

This section covers the evaluation of the OmpSs GPU support component. The experiments analyze the Nanos++ runtime performance when running several OmpSs applications with GPU task offloading.

4.1.5.1 Methodology

In order to evaluate the Nanos++ runtime with GPU support, a set of OmpSs applications were selected and their scalability was measured within the runtime environment.

Environment. The testing system was a multi-GPU system running CentOS 5.3 and it had two Intel Xeon E5440 with four cores each and four Tesla M2050 GPUs, each with 2.62 GB of memory. The total amount of system memory was 15.66 GB.

```
#pragma omp target device (cuda) copy_deps
#pragma omp task inout([ts][ts]tileC) in([ts][ts]tileA, [ts][ts]tileB)
void dgemm_task (double *tileA, double *tileB, double *tileC, int ts)
{
  double alpha = 1.0;
  cublasHandle_t handle = nanos_get_cublas_handle();
  cudaStream_t stream = nanos_get_kernel_execution_stream();
  cublasSetStream(handle, stream);
  cublasDgemm(handle, CUBLAS_OP_T, CUBLAS_OP_T, ts, ts, ts, &alpha,
          tileA, ts, tileB, ts, &alpha, tileC, ts);
}

void matmul (int m, int l, int n, double **A, double **B, double **C, int ts)
{
  int i, j, k;
  for (i = 0;i < m; i++)
    for (j = 0; j < n; j++)
      for (k = 0; k < l; k++)
        dgemm_task (A[i*l+k], B[k*n+j], C[i*n+j], ts);

  #pragma omp taskwait
}
```

Figure 4.7: OmpSs tiled matrix multiply example using the GPU to offload the computation

All the application codes where compiled with the Mercurium compiler with optimization level $-O3$. GCC version 4.3.4 and CUDA version 3.2 were used as back-end compilers for the CPU and GPU parts respectively.

Experiments. The selected applications were run with different configurations of number of GPGPU devices to obtain the performance and scalability of each application. The biggest possible data set was selected so that the performance obtained was not limited due to using a small problem size.

In addition, four applications were selected to run with different configuration parameters of the runtime to evaluate the impact of such parameters as well:

- **Software cache policies**:
  - **No-cache**: It emulates the absence of the Nanos++ software cache by always moving data in and out for each task.
  - **Write through**: Shown as *wt* in the charts, it propagates GPU memory writes to main memory immediately.
  - **Write back**: Shown as *wb* on the charts, it delays the writing to main memory until the last moment (this is forced by a `taskwait`).
- **Runtime scheduling policies**:
  - **Breadth-first scheduler**: Shown as *bf* in the charts, the details of this scheduler can be found in Section 2.1.4.3. It basically does a breadth-first task scheduling, but prioritizes the execution of dependency chains.
  - **Locality-aware scheduler**: Shown as *locality* in the charts, this scheduler is explained in Section 2.1.4.3. In short, it takes into account where data is located to schedule tasks and minimize the amount of transferred data.

- **Overlap of data transfers with computations**: when the overlapping option is activated, it is shown as *ovl* in the charts.

4.1.5.2 Results

This section presents the results of the selected applications run with the different runtime configurations.

Matrix Multiply. The general details of this application are described in Section 2.2.9. In this test, each matrix had 12288×12288 single-precision floating-point elements and was divided into tiles of 1024×1024 elements. The computational tasks called the CUBLAS library to calculate the results.

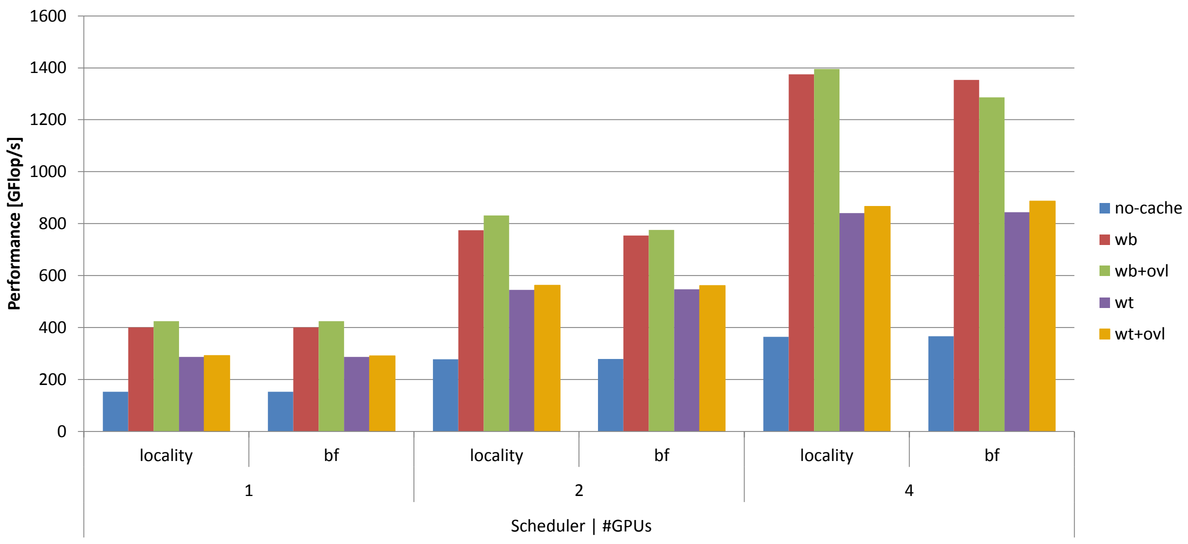


Figure 4.8: Matrix multiply performance results for OmpSs with GPU support

Figure 4.8 shows the evaluation of the matrix multiply. The reduction of data transfers impacts directly on application's performance: when the use of any cache is avoided, the application gets the lowest performance, as data are moved back and forth each time a kernel needs them. Using a write-through policy improves the performance thanks to the data reuse, but writes still create a significant number of transfers that limit application's performance. Using the write-back policy helps to reduce this effect and obtains the best performance of all three policies. In addition, the scheduling policy has also a smaller impact on performance as the breadth-first scheduler can also preserve data locality, but not as good as the locality-aware scheduler does. Finally, enabling the overlap of data transfers with computations gives better performance as well. In terms of scalability, the application scales close to linearly, although there is a small performance loss in the case of running with four GPUs.

STREAM Benchmark. The STREAM benchmark is explained in Section 2.2.8. In this case, the size of each array was 320 MB and they were divided into 16 chunks of 20 MB each. Double-precision floating-point data was used and the benchmark was run for 10 iterations.

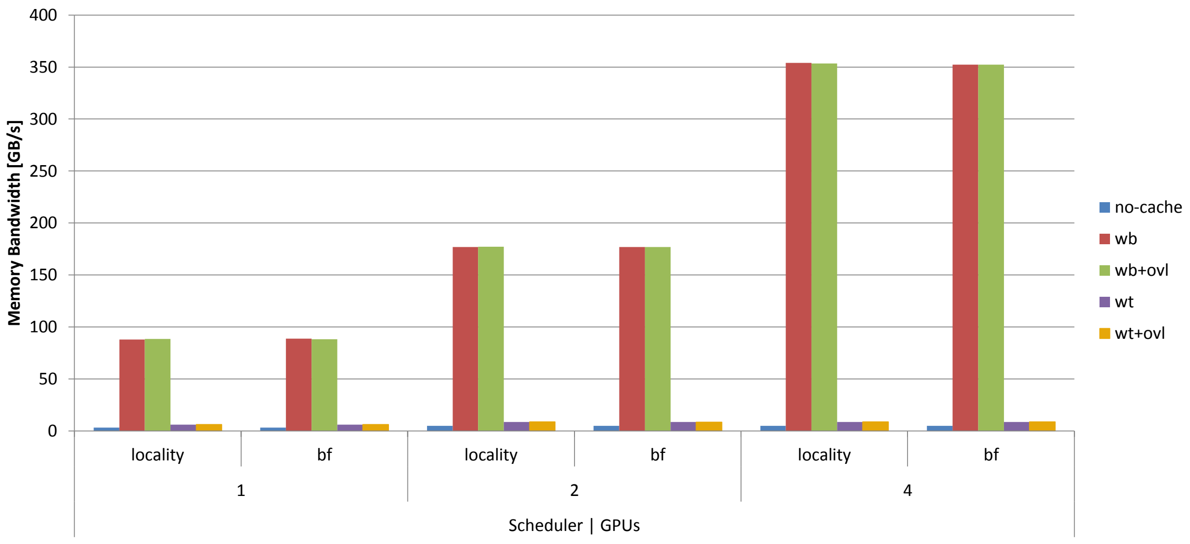


Figure 4.9: STREAM performance results for OmpSs with GPU support

The performance of STREAM benchmark can be found in Figure 4.9. Since the algorithmic structure of this benchmark is very simple, there are no performance differences between the tested schedulers. Nevertheless, the key point of STREAM is the memory management: no-cache and write-through policies move data to main memory every time a task writes to the GPU memory, which overloads the runtime with avoidable memory transfers and has a notably bad effect. In contrast, write-back handles better the data management and obtains a good performance. Enabling overlapping plays a minor role for this benchmark, as its objective is to stress the memory, so there are too few computations to be overlapped with data transfers. The runtime is able to scale almost linearly from one to four GPUs.

N-Body Simulation. The general characteristics of this simulation can be found in Section 2.2.5. The simulation was run for 10 iterations of a system with 122880 bodies.

The performance results of the simulation are presented in Figure 4.10. N-Body uses a lot of GPU memory and requires data to be shared between all the GPU devices. This causes that the no-cache policy outperforms the rest of policies, which completely fill the GPU memory and trigger the runtime cache invalidation mechanisms. Invalidations in the Nanos++ software cache have a considerable overhead, so they delay the writing to main memory. The no-cache policy avoids these situations which are more costly than just keep sending data back and forth to keep the GPU memory free. With this, good scalability is still achieved with two and four GPUs. Since this application is clearly dominated by the amount of data that needs to be exchanged at the end of each iteration, the scheduling policy and the overlapping feature play a minor role and have almost no effect.

Perlin Noise. The Perlin Noise algorithm is described in Section 2.2.7. For this test, an image of 1024 x 1024 pixels was used and 1500 iterations were performed.

The chart in Figure 4.11 represents the number of Mpixels/s processed by the Perlin Noise algorithm. The application stresses memory usage but the amount of computation is very

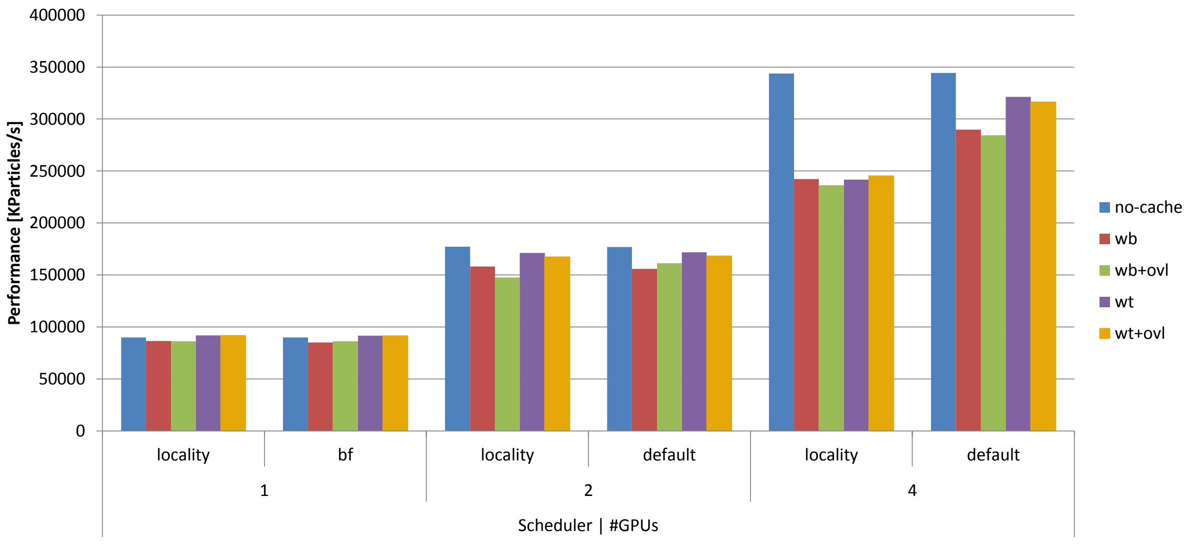


Figure 4.10: N-Body simulation performance results for OmpSs with GPU support

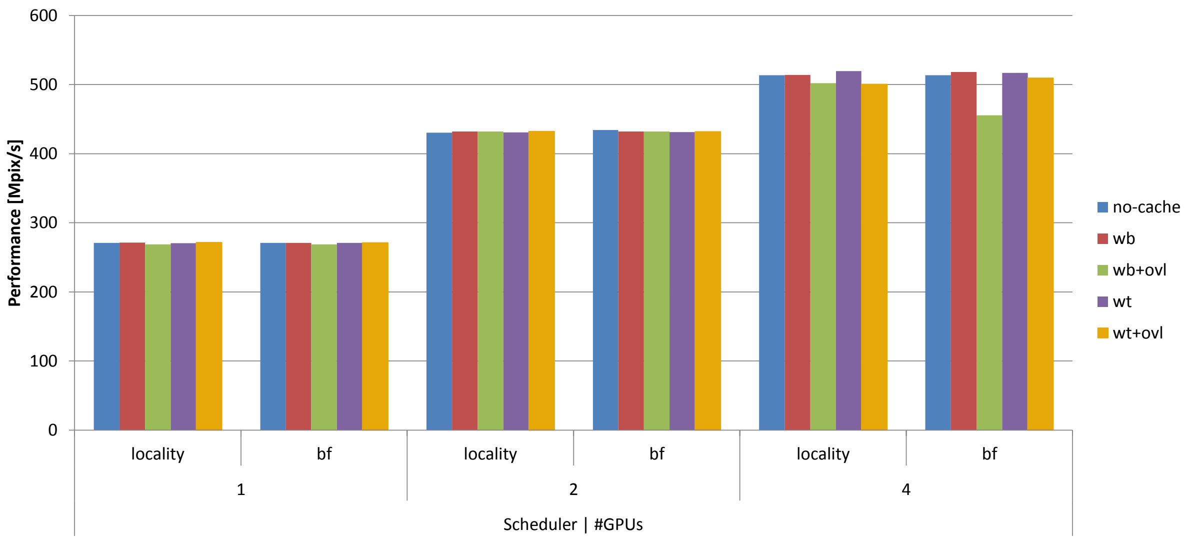


Figure 4.11: Perlin Noise performance results for OmpSs with GPU support

little. Then, most of the execution time is spent in data transfers while the task computation time is almost negligible. Data is not reused between tasks, so no data locality can be exploited. This is why the different configurations tested with cache and scheduling policies and overlapping have no impact in application's performance. The scalability of Perlin Noise is poor as well due to the same reasons.

Black-Scholes. The general explanation of the Black-Scholes benchmark can be found in Section 2.2.1. Arrays of 2^{25} single-precision floating-point elements were used in this case and they were split into 16 chunks each.

Black-Scholes application results for the multi-GPU environment are shown in Figure 4.12. Each task receives a considerable amount of data, so for executions on one and two GPUs the

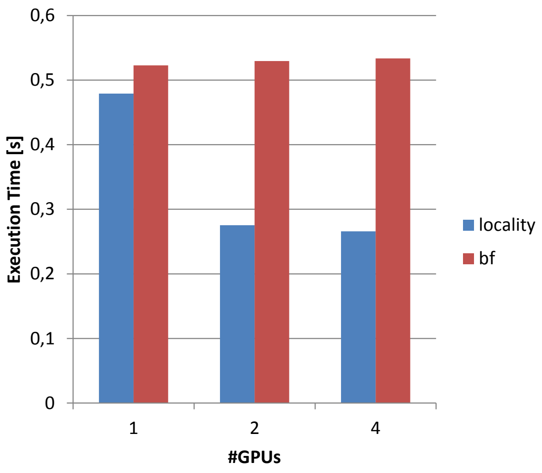


Figure 4.12: Black-Scholes execution time with OmpSs with GPU support

runtime can handle this and scale almost ideally. However, when running the application with four GPUs, the communication bus is collapsed, as the tasks need more data than what the runtime can transfer through the PCIe bus. Using a locality-aware scheduler clearly benefits the performance of this application.

Krist. Krist application is described in Section 2.2.4. This test was run for 100 iterations with 16384 atoms and 65536 reflections.

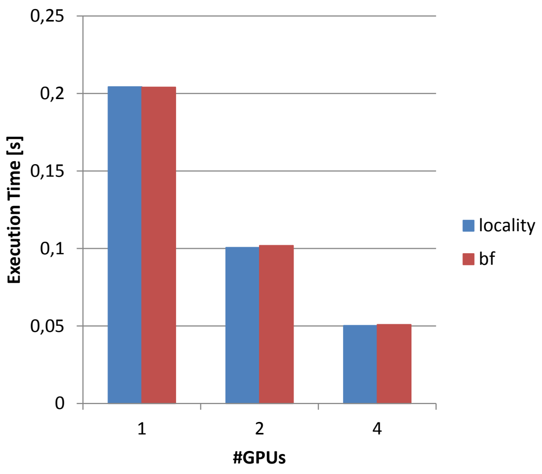


Figure 4.13: Krist execution time with OmpSs with GPU support

Figure 4.13 shows the execution time of the application. It performs very well in the tested environment, as a linear scalability is observed for one, two and four GPUs. In this case, the locality-aware scheduler does not get any extra benefit because the breadth-first policy is able to keep good data locality as well.

4.2 AMA: Asynchronous Management of Accelerators

The fully-asynchronous accelerator support is called AMA: Asynchronous Management of Accelerators. This section describes the AMA design in detail, along with the specific implementations that were done for both GPGPUs and Intel Xeon Phi devices. These implementations and their successful results prove the validity of the AMA design. First, the design of this fully-asynchronous approach is described. Then, the following sections give the implementation details for both the GPU and Xeon Phi support. Finally, the performance results for the two implementations are presented and discussed.

4.2.1 AMA Design

The main goal of the fully-asynchronous approach is to speed-up applications from the runtime side, so that they experience a performance increase while their source code remains untouched. This is the advantage of hiding the particular actions related to accelerators from the programmer point of view: the issue of such actions is done inside the runtime, so it can be changed and optimized as desired, without the application being aware of it.

AMA has been carefully designed to be completely asynchronous. This means that the host side will never block for any device synchronization nor communication. In order to accomplish this objective, AMA establishes that the host-device communication is always done through events and callbacks. Events and callbacks were chosen as the synchronization mechanisms because the most widely-used accelerator programming languages (CUDA, OpenCL) offer such mechanisms to manage asynchronous device operations.

Then, every operation (data transfer or computation offload) will have an associated event that will reflect its status. The host will use such event to query for the operation state. Optionally, an event can have a list of *actions*, i.e. other operations that must be performed once the event's associated operation is finished. Actions can target both host or device and can be either synchronous or asynchronous. Synchronous actions will typically happen on the host and asynchronous actions will usually target the device. The specific device implementation is free to decide whether the action trigger is managed through callbacks or any other equivalent mechanism.

There are three possible states for an event, described below in chronological order:

- **Pending**: Event's operation has been issued to the device, but still not finished.
- **Raised**: The device operation associated to this event is finished.
- **Completed**: All the actions related to this event have been processed. In other words, if the action was synchronous, it has been triggered and finished. If the action was asynchronous, it has been issued, but may or may not have finished: a different event has been created for this new asynchronous operation and such event must be used to query for its status.

4.2.1.1 Event-driven Flow

The execution of tasks on the accelerator mainly follow an event-driven flow. Once the runtime scheduler assigns a task to the accelerator, the following steps are performed:

1. Task active stage begins and hence, input data transfers will be issued. For every transfer, a new event will be created and set to reflect the transfer state[2]. Each event will have at least one action to notify the runtime software cache the completion of the transfer. The last event will also have an additional action that will trigger the task execution. In the case where there is no need to transfer input data (because the task has no input data or because the data is already in the device memory), the task execution can start immediately. It is left to the specific implementations to decide whether this action is started directly or a *false* event with a raised state is created to trigger the task execution.

2. Immediately after this, the architecture-dependent component is ready to request another task to the scheduler. If a new task is assigned to the device, step number 1 can be started for the new task.

3. Once all the input data transfers have finished, the task execution action will be triggered and, thus, the run stage will begin. The specific offloading mechanism of the accelerator will start the task execution on the device and will create a new event to reflect the execution state. This event will have, at least, one action that will trigger the completion stage of the task. Since the architecture-dependent component has already issued input data transfers for the following tasks, the overlapping of input data transfers and computations is very likely to happen.

4. After the task has been executed on the device, the completion stage action will be triggered and thus, the output data transfers will be issued. In addition, the dependency layer will be notified as well for the task completion. Similarly to step number 1, every data transfer will have an associated event to reflect its state and, at least, one action to notify the software cache when the transfer is finished. The last event will have an additional action to release the task-related data structures or perform any clean-up operations needed by the runtime. In this case, if the task does not need to transfer output data (because it does not produce any data or because the software cache decides to delay the data transfers), there is no need to create an extra event and the task-related data structures can be directly released. Since at this point there may be several task cycles initiated, the overlapping of output data transfers with either input data transfers or device computations is very likely to happen.

Figure 4.14 shows the event-driven flow of AMA design: the starting point is to check if there is any raised event. If so, its actions are processed, the event is marked as completed and the process starts again by checking if there is another raised event. If no raised events are found, the architecture dependent component asks the scheduler for another task. If a new task is assigned to the device, task's active stage is started and it checks again for any raised event.

The aim of this flow is to fill the accelerator work queues as much as possible, so that it always has some work to do and is never idle. Hence, we consider that it is more important to give priority to the operation of processing actions of raised events rather than getting a new task for the device: an event changes its status to raised when its device operation is finished, so

[2] The way the event is set to reflect its associated transfer state is implementation-dependent of the accelerator type, so it is left open for the general AMA design.

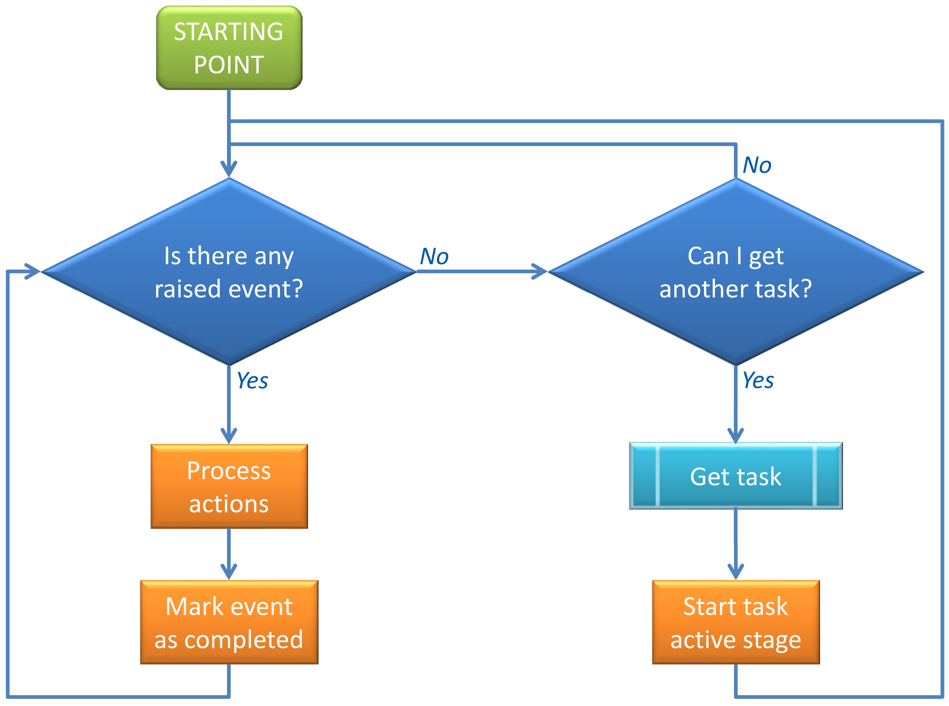


Figure 4.14: Event-driven flow of AMA

there is a probability that the device becomes idle after finishing the operation. By processing the actions of the raised event, the probability of adding more work to the device increases: for example, when the input data transfers are finished, a task will be offloaded to the device. This way, we also give priority to finish the work that has already started on the device rather than getting new work.

This design is ideal for task-based frameworks where the runtime has information about the future tasks that will be run and the data they need: data prefetching can be issued for several tasks ahead so that accelerator work queues have enough operations to overlap and keep the resources busy.

4.2.1.2 FTFR

In OmpSs, there is a restriction regarding task execution order: tasks that are assigned to the same device are run following the same order of assignment. This approach can work well for CPU tasks, but it is not the ideal situation for accelerators. Therefore, AMA adds a simple, small scheduling policy to reorder the execution of tasks that are assigned to the same device, called First-Transferred-First-Run (FTFR). This policy establishes that a task can start its run stage as soon as it has its data available on the device, disregarding the state of other previous tasks. FTFR does not replace the general scheduling policy of the runtime; it is a supplementary policy. The general scheduling policy decides where and when each task is run. FTFR decides the execution order of the tasks that have been assigned to the same device at approximately the same time. This policy is very advantageous for accelerators when tasks that need input data transfers are mixed with tasks that either do not need input data or their input data is already available on the device: tasks with no input data can be

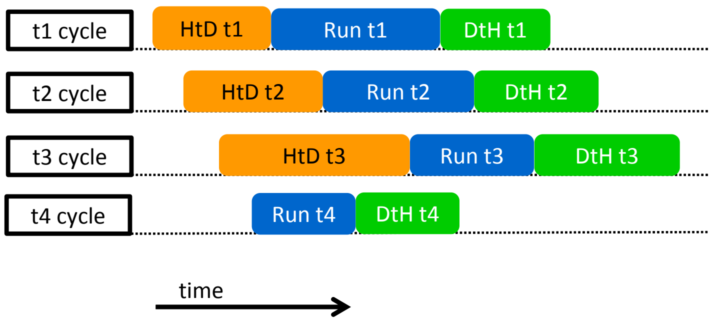


Figure 4.15: Task execution flow with AMA

run directly on the device while tasks with input data wait for their input transfers to finish and both operations can be overlapped.

The execution flow of four tasks *t1*, *t2*, *t3* and *t4* and the potential overlapping of their task stages with AMA is represented in Figure 4.15, following the same convention as Figure 4.1. If supported by the underlying hardware, the active stage of the tasks (with their data transfers) can start one after another, overlapping all the operations. In this case, since *t4* does not need input data transfers, its execution can begin immediately thanks to the FTFR policy, even though it was the last task assigned to the device. Without FTFR, the execution of *t4* would be delayed until the execution of the other three tasks is started. Then, the different stages of the four tasks are completely overlapped with each other and resource utilization is maximized. This illustration shows that AMA is able to overlap all the device operations, but there may be accelerators, limited by the hardware, that cannot support overlapping certain combinations of operations. In these cases, such operations will be delayed until the hardware can issue them, but this is completely compatible with AMA as well.

The AMA's event-driven execution flow can largely speed-up application performance, as the overhead of task management and data transfers is hidden by the execution and data transfers for other tasks. As a result of applying this design, host-side threads are never blocked, so they can do other useful work. This also gives the opportunity of making other runtime components smarter and more powerful, even if they would increase runtime's overhead. Thanks to AMA, several OmpSs scheduling strategies have been improved, as explained in the next chapter.

4.2.2 OmpSs GPU Support with AMA

This section explains the implementation of AMA on top of the OmpSs GPU device support. There are several aspects that remain the same as the semi-asynchronous approach, like the GPU initialization (explained in Section 4.1.2.1) and the GPU memory management (described in Section 4.1.2.2). The host page-locked memory is managed in the same way as well.

4.2.2.1 Event-driven Flow

The two main differences between the semi-asynchronous and the AMA approaches are the way how host and device synchronize and the number of tasks that can be handled simulta-

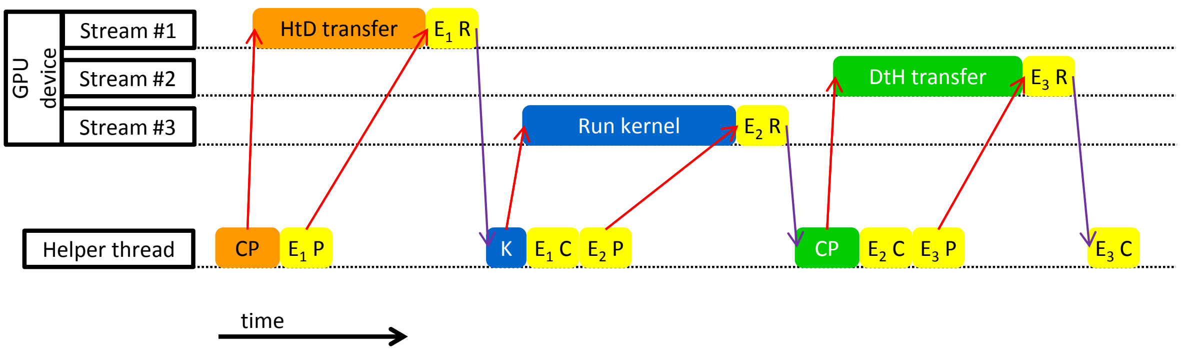


Figure 4.16: Distribution of asynchronous operations and events on GPGPU device streams

neously.

The AMA implementation for GPUs follows an event-polling mechanism: the host-device synchronization is done by inserting a CUDA event after each device asynchronous operation (data transfer or kernel launch). The helper thread registers the CUDA event right after issuing a device operation and creates the appropriate set of actions for that event. Both the operation and the event are issued to the same CUDA stream. According to CUDA specification, the event will not be raised until the preceding asynchronous operation has been completed on the device side, so it will be used to check for the completion of its preceding asynchronous operation. Figure 4.16 shows this mechanism: after each data transfer (CP) or kernel launch (K) from the host, an event (E_1, E_2 and E_3, in yellow) is created with a pending (P) state. When the operation completes on the device side, the event changes the state to raised (R). Then, helper thread processes its actions and marks the event as completed (C). Each device operation is issued to a different stream to overlap with device operations from other tasks.

In this approach, the GPU device helper thread can ask the scheduler for several tasks consecutively. As soon as a new task is assigned to the device, its active stage is started. The thread holds a list of pending events for its device and the state of each event is frequently checked by querying CUDA. When a raised event is found, the thread executes its associated actions. If no raised events are found, the thread requests a new task to the scheduler. This process is repeated over the whole application execution.

In order to favour load balance when more than one processing unit is used, the OmpSs GPU component has a *task prefetching* threshold, which limits the number of tasks that GPU helper threads can handle simultaneously. In other words, when the number of non-completed tasks of the device reaches the threshold, the helper thread is not allowed to request a new task until one of the current tasks is finished. This threshold can be configured through an OmpSs environment variable. Choosing a lower threshold is better to balance the work between processing units, but it can decrease the number of overlapping operations.

Several CUDA streams are used to overlap as many operations as possible. Two CUDA streams are dedicated to data transfers (one for each host-device direction). There is no need to have more than two streams because the hardware limits the number of simultaneous data transfers to one in each host-device direction. However, more streams can be added in the future if the hardware of next-generation GPGPUs supports more simultaneous transfers.

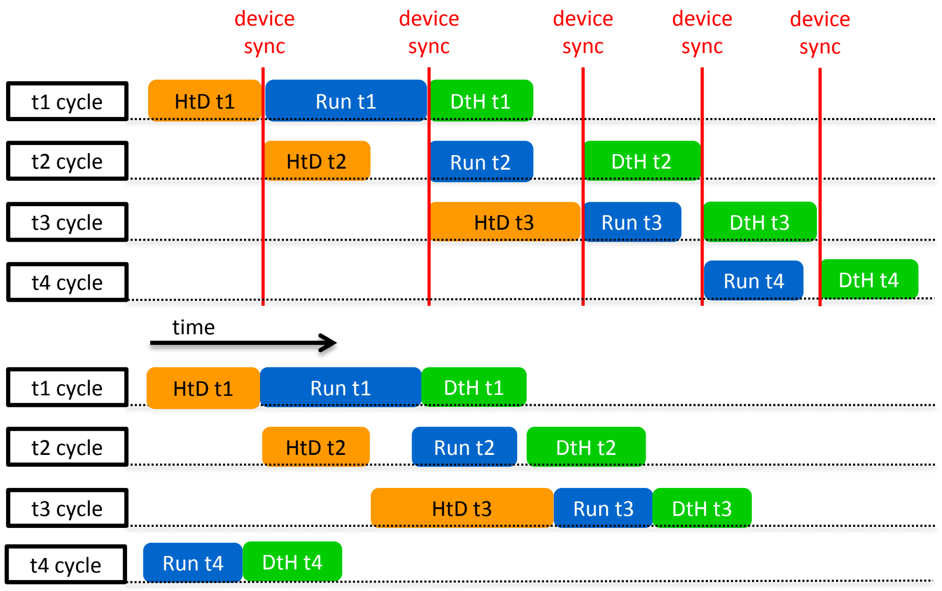


Figure 4.17: Task execution comparison between semi-asynchronous and AMA approaches on a GPGPU device

CUDA's concurrent kernel execution feature is exploited as well by having multiple streams for kernel executions. The number of streams devoted to kernels is set accordingly to the *task prefetching* configuration variable and different streams are used for subsequent task executions in a Round Robin fashion.

Figure 4.17 compares the execution of four tasks *t1*, *t2*, *t3* and *t4* on a GPGPU with the semi-asynchronous approach (at the top) and the AMA implementation (at the bottom). The semi-asynchronous execution takes more time due to the device synchronization stages and the lack of overlapping at several points. In contrast, the global execution time with AMA is lower because the synchronization points have been removed and thus, overlapping time is increased. Still, some gaps are observed due to hardware limitations: a kernel can start only when all thread blocks of all prior kernels from any stream have started and two data transfers in the same direction are serialized. That is why tasks *t1* and *t2* can only partially overlap their execution. Since *t4* task does not need input data transfers, the FTFR scheduler can advance its execution right at the beginning: there is no need to wait for *t1*, *t2* and *t3* data transfers and execution. Then, *t4*'s output data transfers are overlapped with *t1*'s execution and *t2*'s input data transfers. The overall result is that the AMA approach has significantly reduced the total execution time of these tasks. Moreover, *t4*'s dependences would be released at the end of its execution, so its dependent tasks would be ready to run much earlier.

4.2.3 OmpSs Xeon Phi Support with AMA

This section describes how task offloading on Intel Xeon Phi cards in OmpSs is supported by means of the hStreams library [20]. Developed by Intel, this library offers an interface to offload pieces of code on a Xeon Phi device. Conceptually, hStreams is very similar to CUDA or OpenCL: memory transfers must be explicit between host and device memory

```
// Allocate device memory
void * nanos_malloc_hstreams (size_t size);

// Free device memory
void nanos_free_hstreams (void *address);
```

Figure 4.18: Nanos++ API functions for allocating and deallocating Xeon Phi memory

spaces, streams and events are used to issue and control device operations, data transfers and offloaded executions are asynchronous, etc.

The OmpSs Xeon Phi support on top of hStreams has been developed in an iterative and interactive process with Intel: early software releases were provided by Intel, new features have been requested to Intel and several bugs inside the hStreams library have been reported.

We took advantage of class abstraction and inheritance to avoid duplicated code between the GPU and Xeon Phi support components. Then, the main execution flow implementation is shared between both devices and only some small parts have been specialized for each device.

4.2.3.1 Xeon Phi Accelerator Initialization

The hStreams library needs to be initialized before any call to the library and finalized at the end of the application. The initialization includes setting the desired options for the Xeon Phi device, like configuring the number of partitions or OpenMP core thread affinity.

The OmpSs Xeon Phi support component performs all these operations internally, so that they are hidden from the programmer side. The programmer can configure the number of hStreams partitions through an OmpSs environment variable.

Like the GPU component, one helper thread is created for each Xeon Phi card in the system and linked to one of the cards. In this case, the device characteristics are also captured to guarantee a correct execution of the application.

4.2.3.2 Xeon Phi Memory Management

By default, data is allocated on the Xeon Phi memory space the first time they are needed by a task. However, we detected that the hStreams interface used to allocate such data takes a long time to perform data allocations. Thus, for performance reasons, the OmpSs Xeon Phi support offers two Nanos++ API functions to allocate and deallocate user data. Figure 4.18 shows the syntax of these functions. In this case, it is not possible to follow the same approach as the GPU support component because the hStreams streams handle dependencies between operations based on the address of their parameters. If the whole device memory was allocated at once, we could break this hStreams dependence detection mechanism. Figure 4.18 shows the syntax of the functions.

4.2.3.3 Event-driven Flow

The hStreams library provides a slightly different stream abstraction compared to CUDA: operations issued to the same stream may not be executed in a FIFO order. Only those dependent operations, referring to the same host address, are guaranteed to execute in order.

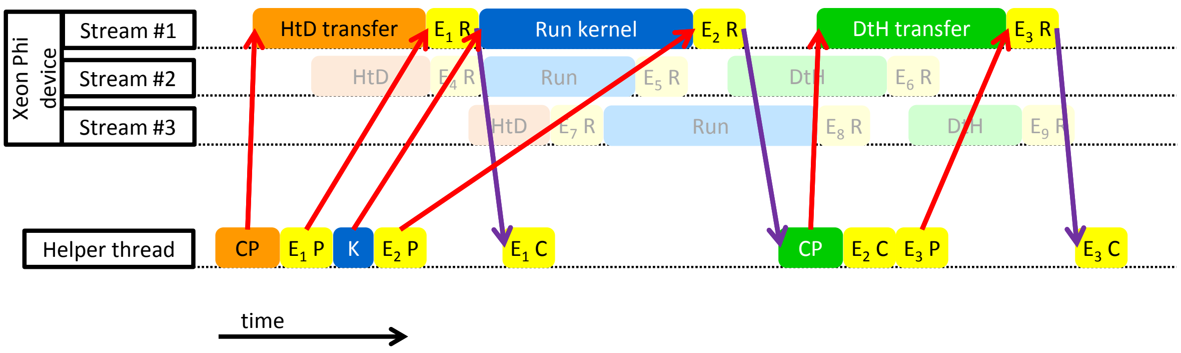


Figure 4.19: Distribution of asynchronous operations and events on Xeon Phi device streams

This automatic-dependence detection allows the OmpSs Xeon Phi support component to apply an optimization in the AMA design: it is not necessary to wait for the completion of input data transfers to launch the kernel. Instead, task offload can be issued immediately after its input data transfers in the same stream and the hStreams library will preserve their dependencies. So, in this case, the three stages of a task (active, run and completion) are issued to the same stream, and several streams are used to overlap the stages of different tasks.

In order to maximize resource utilization, the device helper thread creates several partitions of the Xeon Phi card: cores are evenly distributed between partitions. Then, tasks are assigned to partitions in a Round Robin fashion. In addition, several streams per partition are created, so that there can be several operations overlapping for each partition.

All device operations are issued asynchronously and an hStreams event is associated to each operation. The hStreams asynchronous API functions receive as one of their parameters a pointer to an hStreams event, so the event is automatically associated to its API call. Like CUDA, the library offers calls to either wait for event completion or query its state. The Xeon Phi helper thread always uses the query method to avoid blocking. Figure 4.19 shows how asynchronous operations are issued for a task: the input data transfers and kernel execution are launched one after the other in the same stream. Unlike CUDA, the hStreams runtime creates and initializes the events automatically for each operation. The events associated to input data transfers are still needed to notify the OmpSs software cache of their completion. Finally, when the kernel launch is completed, task's output data transfers are issued. For simplicity, the complete event-driven flow is showed only for one task, issued to *Stream #1*, but the operations of other tasks (shaded boxes) can be handled simultaneously in other streams (in this example, *Stream #2* and *Stream #3*).

Figure 4.20 illustrates the task execution flow of four tasks *t1*, *t2*, *t3* and *t4*. The active stages for each task are issued one after the other. According to hStreams specification, the data transfers happen simultaneously from the programmer point of view. However, it is not clear how the DMA transfers are programmed in the hardware. Assuming that the Xeon Phi device is divided into four partitions, each task runs on a different partition, so they can run in parallel on the same device. Once the tasks are executed, their output data transfers are issued.

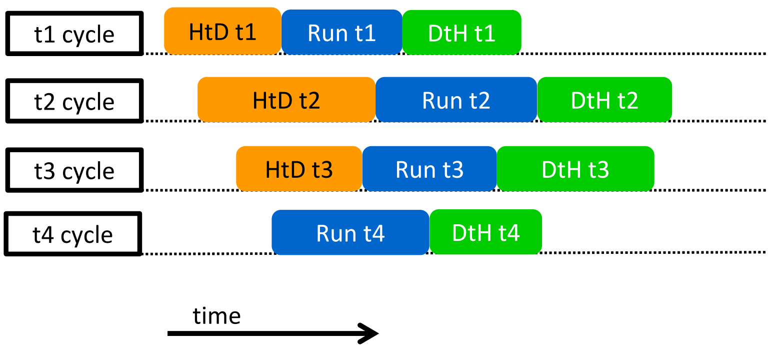


Figure 4.20: Task execution flow on a Xeon Phi device

4.2.4 OmpSs Example Using Intel Xeon Phi

The code presented in Figure 4.21 shows an OmpSs implementation of the tiled matrix multiply algorithm presented in Chapter 2 that offloads the computation to the Xeon Phi. This code looks very similar to the GPU version presented in Section 4.1.4: the same OmpSs directives are used, with just changing the **`device(hstreams)`** clause. The Xeon Phi kernel implementation is provided as well. So, the programmer can easily change the targeting device with small source code modifications. Moreover, tasks targeting different devices can be combined together in the same application.

The OmpSs directives have the same semantics as explained in the GPU example: the task will be run on the Xeon Phi by means of the hStreams library and the OmpSs runtime will manage all the needed data transfers.

In this case, the MKL library is invoked from the kernel code to perform the computation. Like the GPU version, the user is responsible to either write their own kernel code or use an appropriate library. Then, OmpSs takes care of all data movements and kernel synchronizations. Since these operations are always done inside the runtime, the same application can be run in a multi-device environment and different kinds of optimizations can be performed by the OmpSs runtime.

4.2.5 Evaluation of AMA Design

This section presents the performance results of several applications in order to evaluate the AMA design implemented for GPGPU and Xeon Phi accelerators. In these evaluations, the AMA implementations were combined with the *affinity-ready* scheduler and the *priority propagation* mechanism, explained in the next chapter. These scheduling strategies favour task execution on accelerators as well and their implementation was possible thanks to the AMA design. Otherwise, they would introduce too much overhead and it would not be worth applying them.

4.2.5.1 AMA Evaluation on GPGPU Accelerators

The evaluation of AMA for GPGPUs was performed on a multi-GPU Linux system with two Intel Xeon E5-2650 at 2.00 GHz, 62.9 GB of main memory and four nVIDIA Tesla K20c with 2496 CUDA cores and 4.7 GB of memory.

```
#pragma omp target device (hstreams) copy_deps
#pragma omp task inout([ts*ts]tileC) in([ts*ts]tileA, [ts*ts]tileB)
void dgemm_task (double *tileA, double *tileB, double *tileC, int ts)
{
  double alpha = 1.0;
  const char trans = 'N';
  dgemm(&trans, &trans, &ts, &ts, &ts, &alpha, tileA, &ts, tileB, &ts, &alpha, tileC, &ts);
}

void matmul (int m, int l, int n, double **A, double **B, double **C, int ts)
{
  int i, j, k;
  for (i = 0;i < m; i++)
    for (j = 0; j < n; j++)
      for (k = 0; k < l; k++)
        dgemm_task (A[i*l+k], B[k*n+j], C[i*n+j], ts);

  #pragma omp taskwait
}
```

Figure 4.21: OmpSs tiled matrix multiply example using the Xeon Phi to offload the computation

The results were compared with hand-tuned native CUDA versions and the semi-asynchronous approach. The native CUDA codes were compiled with CUDA 5.5 and the OmpSs versions were compiled with OmpSs compiler (using *nvcc* 5.5 and GCC 4.6.4). Optimization level $-O3$ was used in all codes. The same application source code was used with both the semi-asynchronous and AMA implementations. However, the locality-aware scheduler, described in Section 2.1.4.3, was used for the semi-asynchronous approach and the affinity-aware scheduler was used for the AMA approach. The affinity-ready scheduler is part of the contributions of this thesis and is explained in Section 5.2.

The applications were run with different configurations of number of GPU devices and data set sizes to analyze its impact on performance. The performance values shown were computed as the mean value of several executions.

The results obtained from three applications: matrix multiply, Cholesky factorization and N-Body simulation are presented and discussed below.

Matrix Multiply. The general details of this application are described in Section 2.2.9. In this test, the GPU computation is done by calling the *cublasDgemm()* function from CUBLAS library. The different configurations tested are explained in Table 4.1. Double-precision floating-point data was used.

In the native CUDA version, matrices A and C are split into as many chunks as GPUs, so each GPU receives a set of consecutive rows. Matrix B is fully copied to all GPGPU devices. This division avoids data dependences between GPU computations, so all the devices can run their computation part in parallel with the others. Each device calls the *cublasDgemm()* function once with its corresponding chunk of the matrix. The largest data set size that fits in a single GPU's global memory was chosen, so that data transfers between kernel computations are not required.

In the OmpSs version, each matrix is divided into square blocks of 2048×2048 elements. Each task performs a matrix multiply operation on a given block of the destination matrix.

Configuration	*App version*	*Runtime*	*Data set size* [#elements]	*Data transfers accounted?*
Semi-async 16K	OmpSs CUDA	OmpSs GPGPU semi-asynchronous	16384×16384	Yes, all
AMA 16K	OmpSs CUDA	OmpSs GPGPU AMA	16384×16384	Yes, all
Semi-async 32K	OmpSs CUDA	OmpSs GPGPU semi-asynchronous	32768×32768	Yes, all
AMA 32K	OmpSs CUDA	OmpSs GPGPU AMA	32768×32768	Yes, all
CUDA 16K	Native CUDA	CUDA	16384×16384	No

Table 4.1: Matrix multiply configurations

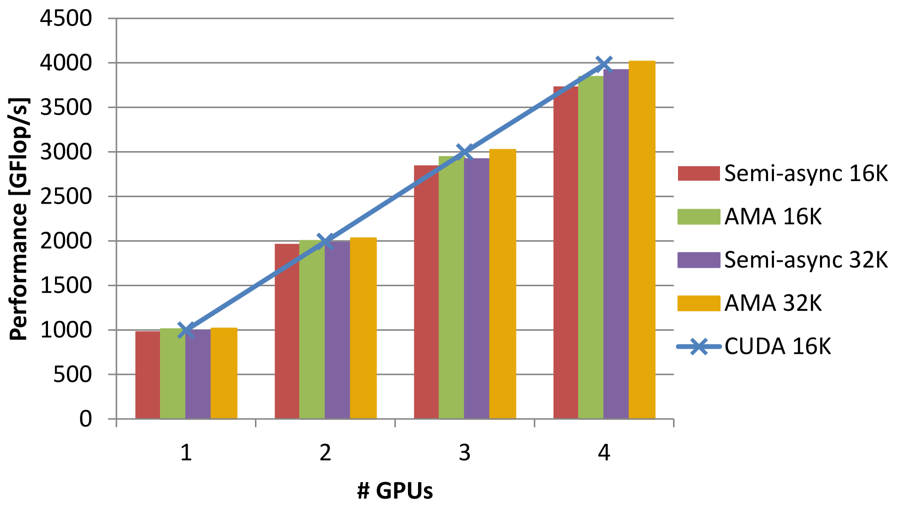


Figure 4.22: Matrix multiply performance results for OmpSs with GPGPU AMA support

Figure 4.22 shows the performance results of matrix multiply run with the different configurations. The results for the *CUDA 16K* configuration do not include the data transfer time; only computation time is accounted. Then, they can be considered as the peak performance of this application, so this is why they are presented as a chart line. Since the algorithm is highly parallel, the performance of *Semi-async 16K* and *Semi-async 32K* get close to the peak values, even though data transfers are taken into account in these measurements. The performance increases as larger data set sizes are used because the application creates more tasks and thus, opens more parallelism. In the case of OmpSs with AMA, a bigger data set size (*AMA 32K*) is also better and the peak performance is achieved. This proves that OmpSs with AMA can fully occupy GPGPU resources and completely hide the overhead of data transfers. The amount of transferred data is optimal in both OmpSs semi-asynchronous and OmpSs with AMA cases, but still OmpSs with AMA gets an extra 4% benefit from the exploitation of the concurrent kernel execution feature.

Configuration	*App version*	*Runtime*	*Data set size* [#elements]	*Data transfers accounted?*
Semi-async 16K	OmpSs CUDA	OmpSs GPGPU semi-asynchronous	16384×16384	Yes, all
AMA 16K	OmpSs CUDA	OmpSs GPGPU AMA	16384×16384	Yes, all
Semi-async 32K	OmpSs CUDA	OmpSs GPGPU semi-asynchronous	32768×32768	Yes, all
AMA 32K	OmpSs CUDA	OmpSs GPGPU AMA	32768×32768	Yes, all
CUDA 16K	Native CUDA	CUDA	16384×16384	Only between devices
CUDA dgemm 16K ceiling ref.	Native CUDA	CUDA	16384×16384	No

Table 4.2: Cholesky factorization configurations

Cholesky Factorization. The Cholesky factorization is explained in Section 2.2.2. Double-precision floating-point data was used for the computation. The four different kernels: *dpotrf*, *dsyrk*, *dgemm* and *dtrsm* were offloaded to the GPGPU. A customized implementation of *dpotrf* based on its corresponding function from MAGMA library was used and CUBLAS library was called for the other kernels. The different configurations used are described in Table 4.2. The block size for all configurations was 2048×2048 elements. Note that *CUDA dgemm 16K ceiling reference* is the performance of matrix multiply previously evaluated in this section. Although the Cholesky factorization cannot scale as good as matrix multiply, these results were added as a reference of scalability.

The CUDA native version used an OpenMP-like fork-join approach due to its complexity of data dependencies between kernels. Data transfers between host and devices, issued at the beginning and at the end of the computation to load data on device memory and get the results back to the host are not taken into account for the performance measurements. However, in this case, GPGPUs need to share data between each other after each join phase of the computation, so these data transfers between device memories are accounted for the performance measurements. The biggest power-of-two data set size that fits in one GPU global memory was chosen, so the whole matrix can be stored in every device memory and additional data transfers and evictions can be avoided during the computation.

In the OmpSs version, each kernel is annotated as a task and task data dependencies are managed by the OmpSs runtime. Different task priorities were used to give more priority to critical tasks. In the OmpSs with AMA configurations, task priorities are propagated up to five levels upwards[3].

Cholesky's performance results are shown in Figure 4.23. The native CUDA version cannot scale across several GPUs due to the synchronization bottlenecks and the fact that GPUs need to exchange data too frequently and these transfers cannot be overlapped with other

[3] The description of this mechanism can be found in Section 5.1.

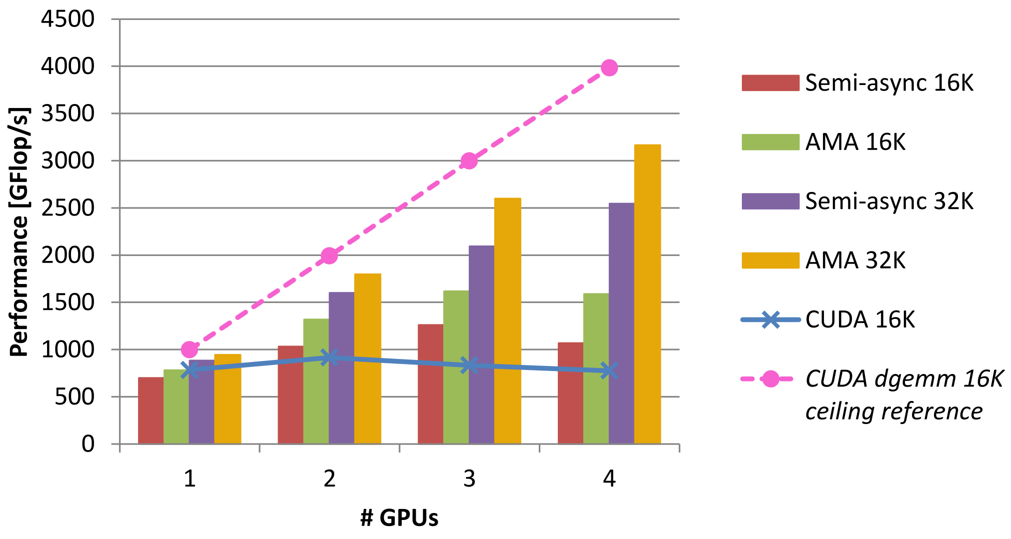


Figure 4.23: Cholesky factorization performance results for OmpSs with GPGPU AMA support

computations. This is the reason why the values of *CUDA dgemm 16K ceiling reference* are shown as well. The two OmpSs configurations with small matrix sizes (*Semi-async 16K* and *AMA 16K*) cannot scale beyond two GPUs because there is a lack of parallelism. In contrast, when the data set size is increased, there is more parallelism available and both *Semi-async 32K* and *AMA 32K* scale better. However, the OmpSs with AMA configurations get better performance than the OmpSs semi-asynchronous executions thanks to its enhanced non-blocking data management and task priority propagation, getting up to 1.5x performance speed-up.

N-Body Simulation. The general characteristics of this simulation can be found in Section 2.2.5. The simulation was run for 10 iterations with different number of bodies using double-precision floating-point data, described in Table 4.3.

The results for the native CUDA version only take into account the data transfers performed between iterations; like in the Cholesky case, the initial and final data transfers between host and devices, to load data and get the result back, are not accounted in the performance results.

Figure 4.24 shows the performance results of running the different configurations of this simulation. The OmpSs semi-asynchronous configurations (*Semi-async 256Kbod* and *Semi-async 512Kbod*) are negatively affected by the amount of data exchanged between iterations, up to the point of not being able to scale across several GPUs. In contrast, the OmpSs with AMA configurations (*AMA 256Kbod* and *AMA 512Kbod*) can scale at the same ratio as the original CUDA application does and, in some points, they even get slightly better performance than the native CUDA implementation. In this case, OmpSs with AMA gets up to 2.2x performance speed-up compared to the OmpSs semi-asynchronous approach.

It is interesting to analyze and compare these results with the first evaluation of the semi-asynchronous model, in Section 4.1.5. In the previous evaluation, the semi-asynchronous

Configuration	*App version*	*Runtime*	*Data set size [#bodies]*	*Data transfers accounted?*
Semi-async 256Kbod	OmpSs CUDA	OmpSs GPGPU semi-asynchronous	262144	Yes, all
AMA 256Kbod	OmpSs CUDA	OmpSs GPGPU AMA	262144	Yes, all
Semi-async 512Kbod	OmpSs CUDA	OmpSs GPGPU semi-asynchronous	524288	Yes, all
AMA 512Kbod	OmpSs CUDA	OmpSs GPGPU AMA	524288	Yes, all
CUDA 256Kbod	Native CUDA	CUDA	262144	Only transfers between iterations
CUDA 512Kbod	Native CUDA	CUDA	524288	Only transfers between iterations

Table 4.3: N-Body simulation configurations

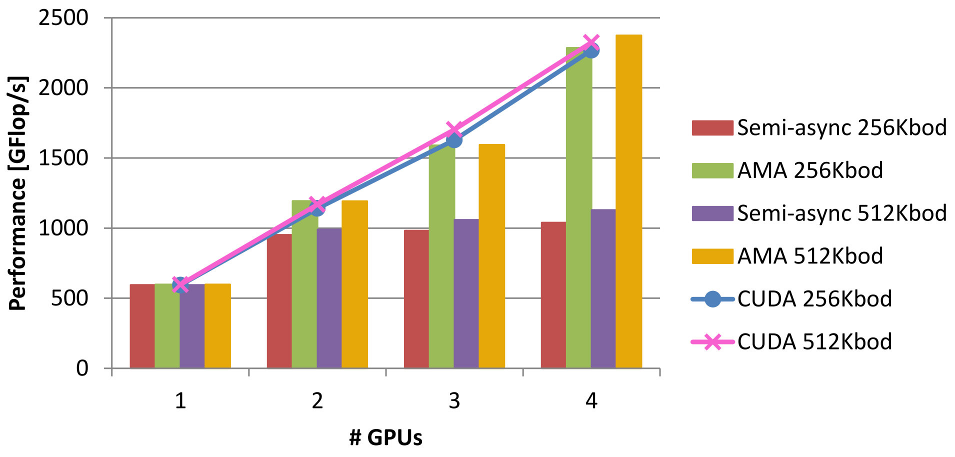


Figure 4.24: N-Body simulation performance results for OmpSs with GPGPU AMA support

model was able to scale a little bit across several GPUs. As new generations of GPGPUs appeared, legacy code does not perform as good as it used to run, up to the point where it becomes obsolete. The N-Body simulation clearly demonstrates two facts: on the one hand, that performance is not guaranteed with new generations of hardware and it must be adapted to fit new characteristics. On the other hand, that it is possible for runtime frameworks to enhance application's performance by internally adapting to new architectures, while the user code remains untouched.

4.2.5.2 AMA Evaluation on Intel Xeon Phi Accelerators

The AMA implementation evaluation for Intel Xeon Phi cards was performed on a system running Linux with an Intel Xeon E5 2x 2680 at 2.6 GHz, 64 GB of memory and an Intel Xeon Phi 7120P card with 61 cores at 1.238 GHz and 16 GB of memory. An early-release package of MPSS version 3.5 was installed in the system.

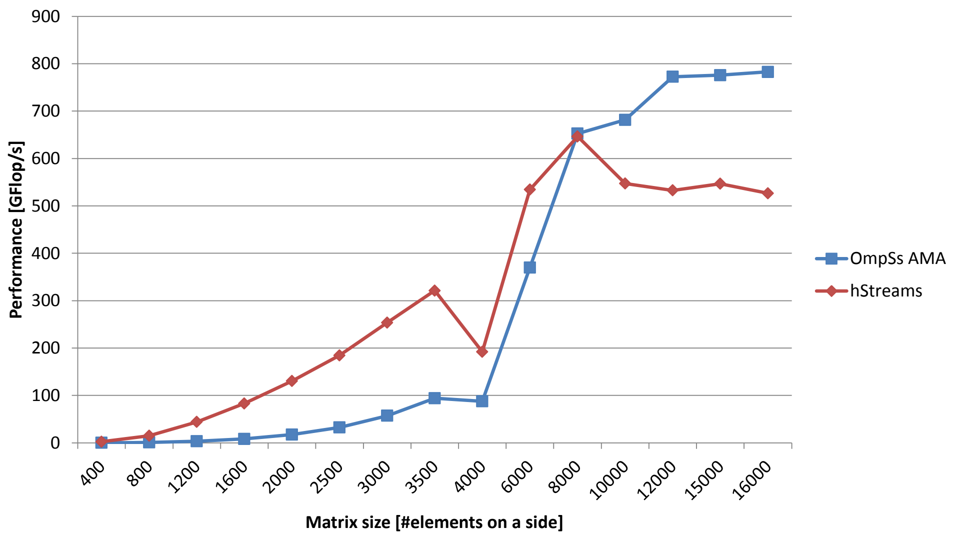


Figure 4.25: Matrix multiply performance results for OmpSs with Xeon Phi support

The results were compared with optimized native hStreams versions, provided by Intel within the hStreams library framework (distributed in the MPSS 3.5 package). The native hStreams codes were compiled with icc 11.1. The OmpSs versions were compiled with OmpSs compiler (using *icc* 11.1 and GCC 4.6.4). Optimization level $-O3$ was used in all codes. The applications were run with different data set sizes to analyze how runtime overheads impact on performance. Results are computed as the mean value of several runs.

The results obtained from two applications: matrix multiply and Cholesky factorization are presented and discussed below.

Matrix Multiply. The general details of this application are described in Section 2.2.9. The tests were run with different data set sizes in double-precision floating-point format. Matrices were always divided into 10×10 square tiles. The Xeon Phi was configured with the number of partitions that gave the best performance results for each version: in this case, four partitions in both versions. The time for all data transfers is accounted in both versions as well.

The hStreams native version is distributed with the hStreams library. The code offloads the computations to Xeon Phi and the native MKL library is called. This implementation uses several streams to overlap data transfers and computations.

The OmpSs code is structured as explained in the general description. OmpSs uses hStreams to offload the computations and the native MKL library is called as well to calculate the result. In this case, the breadth-first scheduler was used, as the executions cannot benefit from any scheduling based on data locality.

The performance results of matrix multiply are presented in Figure 4.25. As the data size is increased, the performance of both hStreams and OmpSs versions increases as well. The overhead of the OmpSs runtime has a significant impact for small matrix sizes, because the computation time is too short. However, as the size is increased the OmpSs performance gets closer to the hStreams native performance, and for big matrix sizes, OmpSs gets even better performance than hStreams. The lack of profiling tools to visualize when data transfers occur and how computations are scheduled on Xeon Phi partitions make performance analysis difficult, as it is not clear to us why the hStreams performance slightly decreases for big matrix sizes. However, we presume that data is not transferred with enough time in advance, so computations are delayed until the data they need is available in device memory. This issue is not observed in the OmpSs version thanks to the AMA design because data transfers are issued much before.

Cholesky Factorization. The Cholesky factorization is explained in Section 2.2.2. Double-precision floating-point data was used to run the computation with different data set sizes. The native Xeon Phi MKL library was used for all the different kernels of the application: dpotrf, dsyrk, dgemm and dtrsm. The matrix was divided into 10×10 square tiles. The Xeon Phi was configured with the number of partitions that gave the best performance results for each version: five partitions in the native hStreams version and six partitions in the OmpSs version. The time for all data transfers is accounted in both versions.

The hStreams native version is distributed with the hStreams library. The native implementation is very complex: many streams are used to overlap computations with data transfers and to preserve data dependencies between the different kernels.

The OmpSs implementation follows the structure described in the general explanation. Task priorities were used in this case and they were propagated up to five levels upwards. The affinity-ready scheduler was used, as it was the only scheduler available with priority propagation support.

The complexity of the native hStreams implementation denotes the difficulty of programming efficient applications for accelerators and still, this code can only run on a single device. In contrast, the OmpSs version is very similar to the sequential CPU algorithm, and only directives (for tasks and barriers) and task implementations (which call the native MKL library and have less than ten lines of code) have been added. Moreover, the same OmpSs version can run on a multi-accelerator system.

Figure 4.26 shows the performance results of the Cholesky factorization. In this case, the differences between OmpSs and hStreams for small data sets is minimal, and as the data set size increases, OmpSs quickly gets better performance than hStreams. Although the native version is highly optimized, OmpSs is able to get more performance thanks to a better dynamic scheduling of tasks and the priority propagation mechanism. As mentioned before, application performance and efficiency cannot be analyzed in detail due to a lack of profiling tools, but we believe that the hStreams version has some load imbalance between Xeon Phi partitions, so it cannot get the same performance as OmpSs.

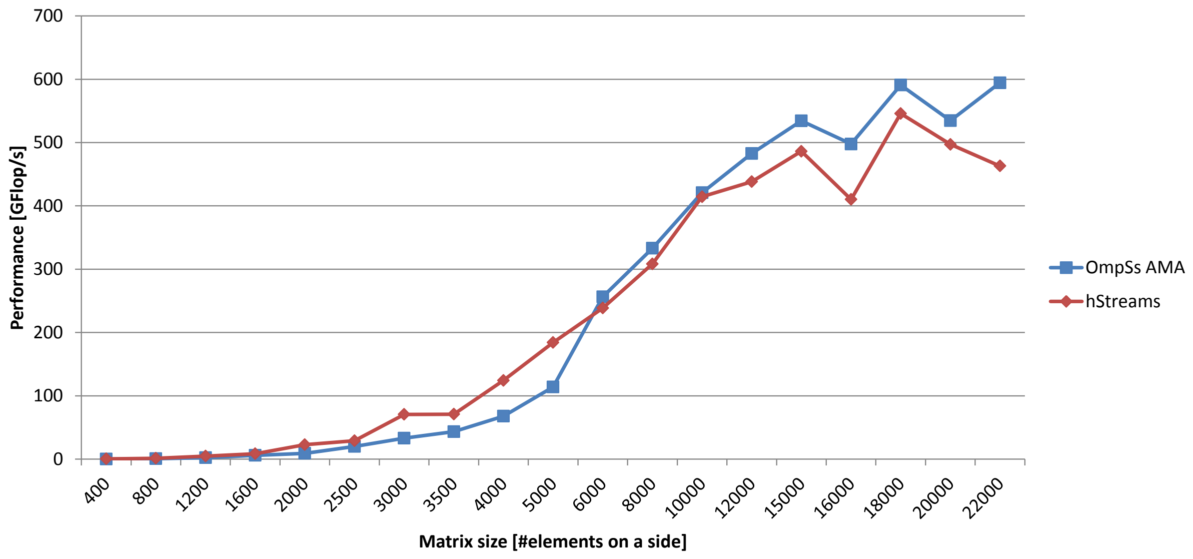


Figure 4.26: Cholesky factorization performance results for OmpSs with Xeon Phi support

Chapter 5
Runtime Scheduling Policies for Heterogeneous Environments

The previous chapter demonstrates that applications targeting accelerators need to be carefully tuned to get good performance and that this accelerator tuning can be transparently applied in runtime frameworks. While this is a key aspect to optimize performance, it is not the only one, as the scheduling of computations is important as well. Making good decisions on the work distribution among computing units and across time can potentially increase application performance.

This chapter describes the thesis contributions related to task scheduling techniques. They have been developed on top of the OmpSs runtime and their main target are accelerator architectures and heterogenous systems. First, two scheduling mechanisms designed for accelerators are presented: the *priority propagation* mechanism and the *affinity-ready* scheduler. And second, two scheduling policies for heterogeneous systems are described: the *versioning* and the *SSMART* schedulers. The use of these last two schedulers show that the optimal performance of an application can only be achieved by the execution cooperation of all the processing units of a system.

5.1 Task Priority Propagation

Generally, in a task-based heterogeneous environment, accelerators consume tasks faster than the host. So, in order to maximize resource utilization and application performance, it is important to give priority to those tasks that open more parallelism (i.e., tasks with several successors) and also to those in the critical path of the data dependency graph. The `priority` clause of the OmpSs programming syntax allows programmers to give a certain priority to each task. Task priority is represented by an integer expression that will be evaluated at application

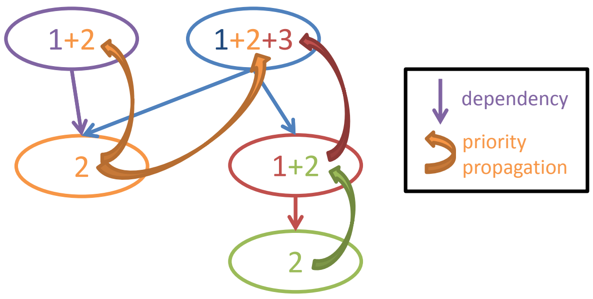


Figure 5.1: Task priority propagation in the OmpSs runtime

run time, when the task is created. Higher integer values correspond to higher priority tasks. Several OmpSs schedulers support task priority and have the ability to propagate priority to the parent task. However, this feature is disabled by default due to its overhead.

Thanks to the AMA design (described in Section 4.2), the idle time of device helper threads can be used to propagate task priorities. Then, the priority propagation overhead is completely hidden and does not impact performance. Moreover, the priority is not only propagated to the parent task, as it is currently supported by OmpSs, but it has also been extended to propagate the priority to several levels of parent tasks. This is not possible without AMA, due to its overhead, but it is particularly useful in applications with complex task graphs, like the Cholesky factorization (explained in Section 2.2.2). In such kind of applications, it is difficult for a task scheduler to find and give priority to the execution of the critical path. With the help of the priority propagation mechanism, paths to critical tasks are found earlier, and thus, its execution is scheduled earlier as well.

Figure 5.1 illustrates how priority is propagated across the task dependency graph: each node represents a task; straight, thin arrows represent dependencies between tasks and thick, curved arrows indicate the priority propagation direction. Each node is labelled with a number that corresponds to the task priority. After the priority propagation, the priority of each task has been increased by its children priority, which, in turn, their priority was also increased by their children priority.

The implementation uses a double-linked data dependency graph so that successor tasks can access their predecessors to update their priority. The construction of this double-linked graph was disabled by default in OmpSs due to its overhead and the navigation was only allowed from parent to child tasks. As mentioned before, the overhead added by the double-linked graph construction and navigation is hidden by the execution of other tasks and their data transfers, so, effectively, it is cost-free in the AMA implementation.

The priority propagation mechanism is not considered to be a scheduler itself, but it is a tool that schedulers can use to enhance their task scheduling quality to ensure that the dependencies of higher priority tasks are satisfied as soon as possible. This mechanism is activated each time a new task is added to the dependency graph, and will affect its predecessors. The optimum number of predecessor levels to navigate and update priority is application-dependent. Then, the scheduling policies that support this feature offer a configuration option to set the maximum number of predecessor levels where priority must be propagated.

5.2 Affinity-ready Scheduler

The use of accelerators usually implies dealing with more than one memory space, even in single-node environments. Consequently, data movements must be performed to device local memory before and/or after device computations. Memory transfers can take a considerable amount of time with respect to accelerator code execution time, so, ideally, they should be minimized.

This is the objective of the existing OmpSs locality-aware scheduler: a task *affinity score* is computed for every memory space in the system. The affinity score takes into account task data size and is computed at task creation time because it has a considerable overhead: if it was computed at the ready stage, it would delay task execution and negatively affect performance. This approach has a clear weakness: the affinity score is likely to be out of date at the ready stage, when the task is going to be run on the assigned processing unit. Then, the objective of reducing memory transfers is not completely fulfilled.

The *affinity-ready* scheduler, which is a contribution of this thesis, solves the problems that the locality-aware scheduler presents. On the one hand, the computation of the affinity score has been modified to better fit accelerator systems. On the other hand, the affinity score is calculated at the ready stage of the task, and the overhead that this may introduce can be completely hidden by the AMA design.

In addition to the global ready queue of the Nanos++ runtime, the affinity-ready scheduler creates a separate ready queue for each memory space of the system. So, processing units fetch tasks from the local queue corresponding to their memory space. When a task becomes ready, the scheduler decides which memory space is the most suitable to place the task. The affinity score is computed as the number of bytes that must be transferred to each memory space in order to execute the task. This information comes from the copy clauses inserted by the user in the application code. Both input and output data transfers are accounted, and *inout* data are accounted twice. Then, the task will be placed in the local queue of the memory space with the lowest score. In case of a tie between memory spaces, tasks are placed in the local queue with the lowest number of tasks. Only the memory spaces where there is a processing unit that can run the task are considered in the scheduling decision. For example, only GPGPU memory spaces will be considered for a task targeting a GPGPU architecture.

The scheduler implements a task stealing mechanism to correct load imbalance. This mechanism also tries to preserve data locality as much as possible and requires the minimum number of data transfers. So, when a processing unit tries to fetch a task from its local queue and the queue is empty, it will try to steal a task from another memory space. The stealing process is done conforming to a set of rules: if the scheduler does not find a task that fits the first one, it will try to find a task that fits the next rules successively, in the following order:

1. There is no need to perform any memory transfer to run the task.
2. Moving the task to its memory space will not trigger a data invalidation in another memory space (only output and *inout* data can trigger invalidations).
3. There is enough free space in the local memory to bring the data needed by the task.

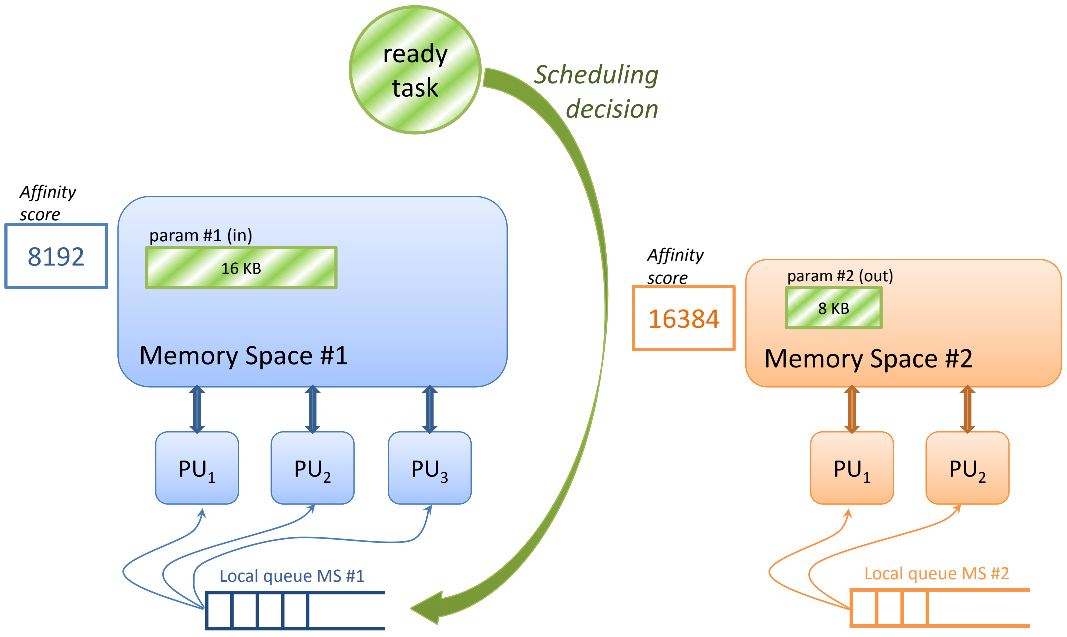


Figure 5.2: Affinity-ready task scheduling in the OmpSs runtime

4. If none of the rules above applied, try to steal a task from any queue, in a Round Robin fashion.

Figure 5.2 illustrates the scheduling decision when a task becomes ready. The system has two different memory spaces with three and two processing units (PU) respectively. The processing units access their corresponding local queues to get tasks. In this example, the new ready task receives two parameters: one input parameter of 16 KB and one output parameter of 8 KB. The input parameter is already present in the *Memory Space #1* and the output parameter is present in the *Memory Space #2*. Then, the amount of additional data needed in *Memory Space #1* is 8 KB (8192 bytes) and this is its affinity score. In the same way, the affinity score of *Memory Space #2* is 16384 (16 KB). Thus, the affinity-ready scheduler decides to place the task to the local queue of *Memory Space #1* because it needs less data to be transferred.

5.3 Versioning Scheduler

Running applications on accelerators can significantly speed-up their performance, but it may be even better not to just use accelerators, but all the computing units of the system as well. The main challenge of this approach is programmability: first, application code must be split into smaller parts; second, the programmer has to decide which processing unit executes each part; and finally, these code fragments probably need additional communication and synchronization between them. OmpSs supports running the different tasks of an application among all the processing units of a system, even if they are of different types. However, the model expects that each task targets one single architecture type. The *versioning scheduler* offers a new feature to OmpSs: the ability to address *heterogeneous tasks*, i.e. tasks that target more than one architecture. Heterogeneous tasks can have a single implementation that targets several architectures, a specialized implementation for each architecture they target or a combination of both.

The objective of this new feature is to provide code performance at low-cost maintenance: as new architectures appear, new programming paradigms do too and applications may become obsolete in a relatively short amount of time. The versioning scheduler offers the ability to join separate pieces of code (new task implementations) to the original application without having to modify it. Programmers can then rewrite certain parts of the old application to improve performance, fit new architectures or adjust to user needs and join these rewritten parts to the original code, just like a puzzle.

The versioning scheduler is able to dynamically choose the most appropriate task implementation each time a task must be executed. As tasks are executed, the scheduler learns and keeps track of them so that it can make more accurate decisions in the immediate future of application's execution. The key point of the scheduler is to improve application's performance by means of resource and thread cooperation.

The details of the versioning scheduler, the source code syntax and the compiler support needed are explained in the following sections.

5.3.1 Syntax and Compiler Support

Application tasks must be annotated in a certain way so that the runtime recognizes heterogeneous tasks and their implementations. Figure 5.3 shows an example of a source code with a heterogeneous task with three task implementations. The main version is called *main_impl* and is annotated like a regular task. The other two versions are called *another_impl* and *yet_another_one* and have an additional **`implements`** clause. Note that all versions receive exactly the same parameters and have the same dependency clauses. The **`implements`** clause is only valid for functions annotated as tasks (it cannot be used in inlined tasks) and its argument always references the main implementation. It is not possible to create an implementation of a task that already has the **`implements`** clause: only regular tasks can have alternative implementations (e.g. *yet_another_one* task cannot be an implementation of *another_impl*). There are no restrictions about task **`device`** clause: the main implementation does not need to target SMP architectures, the programmer can give more than one implementation for each device or even the same implementation can be targeted to more than one device (provided that all devices specified in the **`device`** clause are able to execute the code). A heterogeneous task is then the set of all its task implementations. For each heterogeneous task, the compiler creates a structure with the necessary information for the runtime to recognize the heterogeneous task with all its implementations. Basically, this structure contains a list of devices where the task can be executed and a pointer to the corresponding implementation function address for each device. The distinction between the main implementation and the others is just a compiler requirement and does not affect the versioning scheduler, as all task versions are treated equally at run time.

5.3.2 Data Collection

The versioning scheduler keeps and updates several data structures over the whole execution of the application. These structures are used to collect information related to each heterogeneous

```
#pragma omp target device (cuda) copy_deps
#pragma omp task in ([N]A) out ([N]B) inout ([N]C)
void main_impl (int N, double *A, double *B, double *C)
{
  // Task code
}

#pragma omp target device (cuda) implements (main_impl) copy_deps
#pragma omp task in ([N]A) out ([N]B) inout ([N]C)
void another_impl (int N, double *A, double *B, double *C)
{
  // Task code
}

#pragma omp target device (cuda) implements (main_impl) copy_deps
#pragma omp task in ([N]A) out ([N]B) inout ([N]C)
void yet_another_one (int N, double *A, double *B, double *C)
{
  // Task code
}
```

Figure 5.3: Example of different task versions

HeterogeneousTask	DataSetSize	<VersionId, ExecTime, #Exec>
het-task1	2 MB	<het-task1-v1, 30ms, 200> <het-task1-v2, 18ms, 350> <het-task1-v3, 25ms, 230>
	3 MB	<het-task1-v1, 45ms, 80> <het-task1-v2, 25ms, 300> <het-task1-v3, 40ms, 120>
het-task2	5 MB	<het-task2-v1, 15ms, 40> <het-task2-v2, 20ms, 3>

Table 5.1: Versioning's data structure for heterogeneous tasks

task as shown in Table 5.1. The information is grouped according to heterogeneous tasks and the data set size of each task call, because the execution time of each task is likely to be related to the amount of data that it needs. Then, for each pair of heterogeneous task and data set size, the scheduler stores the mean execution time (*ExecTime*) and the number of executions (*#Exec*) of each task version(*VersionId*). There are two different heterogeneous tasks in Table 5.1: *het-task1* (with three different implementations) and *het-task2* (with two different implementations). In the case of *het-task1*, there are two different groups of data set sizes, because this task has been called with two different data set sizes. For each size, the three implementations have been run several times and their mean execution time has been recorded. Similarly, information for all the tasks implementations of *het-task2* has also been recorded, but in this case, there is only one group of data set sizes, because this type of task has always been called with the same data set size.

Each OmpSs worker[1] thread is currently devoted to run tasks on a single device (SMP, GPU, etc.) and there can be as many workers as machine resources (cores, GPGPUs, etc.). With the versioning scheduler, each worker has its own private task queue. Each element of the

[1] For simplicity, OmpSs *worker* refers to either worker and helper threads, as the versioning scheduler makes no distinction between them.

queue is a pointer to a task that will be run by the corresponding thread. So, it will be used at runtime to assign tasks to threads and keep track of the amount of work each thread has in order to balance the task execution among all threads.

5.3.3 Runtime Scheduling Policy

The versioning scheduler is based on the Nanos++ breadth-first policy, which follows task dependency chains in order to promote data locality and minimize data transfers in a fast and simple way. Before explaining the versioning scheduling policy, several scheduler-related concepts must be defined:

- ❏ **Mean execution time of a task version:** Each time a task is run, its execution time is recorded and its mean execution time is updated as the arithmetic mean of all the task executions. This value is used by the scheduler as the estimated execution time of that task version for future executions.
- ❏ **Fastest executor of a task:** For each group of data set sizes, this is the fastest task version. This concept refers only to those OmpSs workers that can run such task versions.
- ❏ **Earliest executor of a task:** This is the OmpSs worker that can finish the execution of a task version at the earliest time. It will usually be the fastest executor, but in some cases, when the fastest executor is busy running other tasks, the earliest executor may be another OmpSs worker.
- ❏ **OmpSs worker estimated busy time:** Time estimation for an OmpSs worker to finish the execution of all its assigned tasks. It is computed as the addition of the estimated execution time for each task in its queue.

When using the versioning scheduler, the execution of an application is divided into two different phases from the scheduler's point of view: the *initial learning phase* and the *reliable information phase*. The initial learning phase finishes when the scheduler has enough reliable information about the execution of heterogeneous tasks.

The initial learning phase consists of picking task versions from ready heterogeneous tasks in a Round Robin fashion and distributing them among the OmpSs workers. For each task version run, its execution time is recorded and thus data structures of Table 5.1 are filled with this information. The scheduler forces each task version to be run at least λ times during the initial learning phase[2]. Once all tasks versions belonging to the same group of data set sizes have been run at least λ times, the scheduler considers that has enough reliable information and it switches to the reliable information phase[3] for the given group of data set size. This means that the scheduler can have different criteria for the tasks that picks from the global ready queue, depending on whether their corresponding group of data set size has enough reliable information or not.

[2] This threshold can be configured by the user.

[3] Changing from one phase to the other just means that the scheduler changes the criteria to assign task versions to workers.

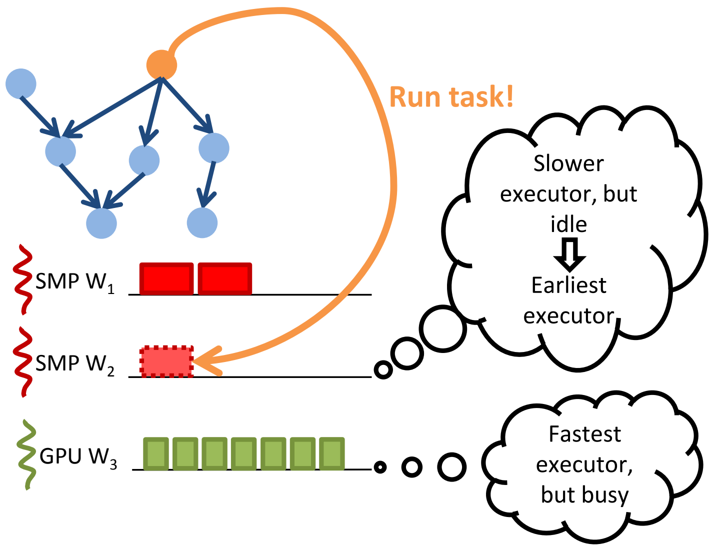


Figure 5.4: Example of a scheduling decision by versioning

During the reliable information phase, the scheduler assigns each task version to its earliest executor (taking into account task's data set size). To make this decision, it takes into account who is the fastest executor of the corresponding group of data set size, but also how busy each worker is (by checking each worker's task list). Figure 5.4 represents an example of this situation: for simplicity, there is only one type of heterogeneous task in the task graph. Each rectangle represents a task assigned to a worker and its width represents the mean execution time of the task version. The scheduler picks the ready task colored as orange and decides which worker should run this task. The *GPU W_3* is the fastest executor of that task (GPU version runs faster than SMP version), but it is busy because it has many tasks in its task list, so the current task would have to wait until all the previous tasks are finished[4]. Although the SMP version is slower, the *SMP W_2* is idle and, in fact, can finish the execution of the current task before the *GPU W_3* has run all the tasks in its queue. The scheduler will then assign the current task to *SMP W_2* because it is the earliest executor.

In the reliable phase, execution information is also recorded exactly in the same way as the previous phase: for each task version, its execution time is computed and its corresponding mean execution time is updated accordingly, so, in a sense, the scheduler is always learning and recording execution information. This makes the scheduler more flexible and easily adapts to application's behavior, even if it changes over the whole execution.

Figure 5.5 synthesizes versioning decision's flow. The starting point is a worker thread getting a task from the ready queue. If the scheduler is in the learning phase for such task and there is not enough reliable information for thread's device, versioning assigns this task to the thread. Otherwise, the scheduler is in the reliable information phase for that task. Then, if the thread is the earliest executor, it will get the task. As an optimization, the thread can still get the task, even not being the fastest executor, if there are more than T tasks in the dependency graph. T represents a certain threshold and is used to determine the amount of work pending

[4] This information is just an estimation based on the past history of each task version, but it is very likely that future executions of the task version will behave similarly to the past executions of the same task version.

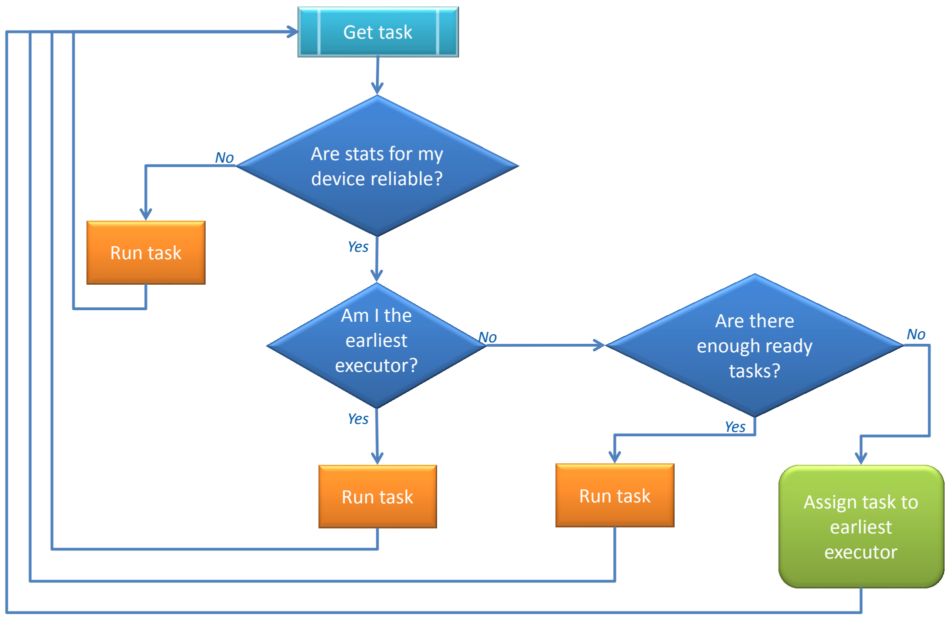

Get task
Are stats for my device reliable?
No
Run task
Yes
Am I the earliest executor?
No
Are there enough ready tasks?
No
Yes
Yes
Run task
Run task
Assign task to earliest executor

Figure 5.5: Versioning scheduler flow

to execute. This optimization is based on the following reasoning: if there is still a lot of work to do and the thread is idle, let it run the task, even if it is not the earliest executor. When there is a few work pending, only the fastest executor can run the task; otherwise, slower executors could harm application performance. This allows the scheduler to use idle units but prevent slow units to execute tasks at later stages of the application that would impact directly on its execution time. Finally, if none of the previous attempts to get the task succeeded, the task will be assigned to the earliest executor.

When the data set size of a ready task differs from what the scheduler has registered in its previous executions, it is considered like a new group of data set sizes and it switches back to the initial learning phase behavior until it has again enough reliable information to move forward to the next phase (the reliable information phase). The versioning scheduler does not allow work stealing between worker threads, so when a task is assigned to a worker, it is not possible to change this decision.

The learning costs of the versioning scheduler are very application-dependent. Although the scheduler never stops learning (because it is always updating the data structures of heterogeneous tasks), we could say that there is an initial learning phase while one or more versions of a task have not been run enough times. But still, the cost of this initial learning phase is application-dependent: for example, if one of the task versions is much slower than the others, the impact of the learning cost will be higher. In the same way, a short run with a few task instances (e.g. less than 10) will be highly affected by the learning costs (applications with 50-100 or more task instances have low learning costs).

5.3.4 Example

Considering the matrix multiply example presented in Figure 2.3 from Chapter 2, Figure 5.6 shows the additional code that should be added to the original matrix multiply code in order

```
// Additional code
#pragma omp target device(cuda) copy_deps implements(dgemm)
#pragma omp task inout([ts*ts]tileC) in([ts*ts]tileA, [ts*ts]tileB)
void dgemm_cublas(double *tileA, double *tileB, double *tileC, int ts)
{
  double alpha = 1.0;
  cublasHandle_t handle = nanos_get_cublas_handle();
  cudaStream_t stream = nanos_get_kernel_execution_stream();
  cublasSetStream(handle, stream);
  cublasDgemm(handle, CUBLAS_OP_T, CUBLAS_OP_T, ts, ts, ts, &alpha,
      tileA, ts, tileB, ts, &alpha, tileC, ts);
}

// Original code
#pragma omp task inout([ts*ts]tileC) in([ts*ts]tileA, [ts*ts]tileB)
void dgemm (double *tileA, double *tileB, double *tileC, int ts)
{
  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, ts, ts, ts, 1.0, tileA, ts,
      tileB, ts, 1.0, tileC, ts);
}

void matmul (int m, int l, int n, double **A, double **B, double **C, int ts)
{
  int i, j, k;
  for (i = 0;i < m; i++)
    for (j = 0; j < n; j++)
      for (k = 0; k < l; k++)
        dgemm(A[i*l+k], B[k*n+j], C[i*n+j], ts);

  #pragma omp taskwait
}
```

Figure 5.6: OmpSs Matrix Multiply task calling CUBLAS

```
#pragma omp target device(cuda) copy_deps implements(dgemm)
#pragma omp task inout([ts*ts]tileC) in([ts*ts]tileA, [ts*ts]tileB)
void dgemm_cuda(double *tileA, double *tileB, double *tileC, int ts)
{
  dim3 block(16, 16);
  dim3 grid(N/block.x, N/block.y);

  dgemm_kernel<<<grid, block>>> (tileA, tileB, tileC, ts);
}
```

Figure 5.7: OmpSs Matrix Multiply task calling a hand-coded CUDA kernel

to have a heterogeneous task with two implementations: the main implementation would be run on a CPU and the additional given implementation would run on a GPGPU. For readability, the original code is also included in this figure.

So, by adding just the piece of code shown in Figure 5.6, Nanos++ with the versioning scheduler will test the two implementations of *dgemm* task and would choose the best version each time the task is called. There is not a hard limit on the number of task versions, so as many task versions as desired can be added. Additionally, Figure 5.7 shows another example of a CUDA implementation of *dgemm* task. This implementation configures[5] and calls a hand-coded kernel called *dgemm_kernel*. This piece of code can be added to the previous matrix multiply application and provide the runtime with three task implementations: the main one for SMP architectures (i.e. general purpose CPUs) and the other two for GPGPU architectures.

[5] Note that no streams are used in the kernel call: when compiling this code, Mecurium compiler will insert the appropriate kernel configuration to use Nanos++ CUDA streams.

5.3.5 Evaluation

The following sections cover the evaluation of the versioning scheduler running OmpSs applications with heterogeneous tasks specialized for CPU and GPGPU architectures.

5.3.5.1 Methodology

A set of three OmpSs applications was selected to measure their scalability and performance with the versioning scheduler, comparing it to the other existing schedulers of the runtime environment.

Additionally, and with the purpose of having a better understanding of the results, the amount of transferred data between host and device memory is also measured over the whole execution of the application. It may happen that the amount of transferred data is much bigger than the total size of application's data, because data may be shared between processing units of different memory spaces. The amount of transferred data is classified into three categories:

- ❑ **Device Tx**: When using two GPGPU devices, it refers to the amount of data transferred between these devices.
- ❑ **Output Tx**: It is the total amount of data transferred from all GPU memory spaces to the host memory space (main memory).
- ❑ **Input Tx**: It is the total amount of transferred data from the host memory space (main memory) to any of the GPGPU devices. If a piece of data is transferred to two different devices, both transfers are taken into account.

Finally, the number of times each task implementation was run is also counted for the versioning scheduler. This gives an idea of how CPU and GPGPU devices cooperate together with application's execution.

Environment. The experiments were run on one node of the MinoTauro cluster at the Barcelona Supercomputing Center. The system runs Linux and each node has two Intel Xeon E5649 6-Core at 2.53 GHz and two nVIDIA GPUs M2090 with 512 CUDA cores. The total amount of main memory for a node is 24 GB. Each GPGPU has 6 GB of global memory.

All the codes were compiled with OmpSs compiler version 1.3.5.8 with optimization level $-O3$. GCC version 4.4.4 and CUDA version 4.0 were used as back-end compilers for CPU and GPU codes respectively. OmpSs runtime version 0.7a with semi-asynchronous GPU support (described in Section 4.1) was used, configured to overlap data transfers with computations.

Experiments. The selected applications were run with different configurations of number of cores and GPGPU devices to obtain the performance of each application. We used as many OmpSs SMP worker threads as cores, placing one worker on each core. For each application, the charts show the results of running the regular version of the application (where each annotated task of the source code is targeting only one device) with the heterogeneous version of the application (where annotated heterogeneous tasks have one or more implementations for different devices).

For the regular version of the application, the fastest combinations of task implementations were chosen, and this is used to evaluate the quality of the versioning scheduler.

For the heterogeneous version, several CPU and GPU task implementations are provided and the versioning scheduler makes the decisions at run time.

Three different OmpSs schedulers were used to compare the results:

- **Breadth-first scheduler**: Shown as *bf* in the charts, the details of this scheduler can be found in Section 2.1.4.3. It basically does a breadth-first task scheduling, but prioritizes the execution of dependency chains.
- **Locality-aware scheduler**: Shown as *locality* in the charts, this scheduler is explained in Section 2.1.4.3. In short, it takes into account where data is located to schedule tasks and minimize the amount of transferred data.
- **Versioning scheduler**: The scheduler presented in this section as a thesis contribution.

In the OmpSs version 0.7a (the latest version by the time these tests were performed), the breadth-first and locality-aware schedulers did not support having more than one implementation for a task, so the heterogeneous version of applications could only be run using the versioning scheduler.

5.3.5.2 Results

This section presents and compares the results of three applications: matrix multiply, Cholesky factorization and PBPI run with the three different scheduling policies mentioned before. It also evaluates how the number of resources impacts application's performance.

Matrix Multiply. The general details of this application are described in Section 2.2.9. In this test, each matrix had 16384×16384 double-precision floating-point elements and was divided into tiles of 1024×1024 elements. Three different task implementations were provided to do this computation: two GPU tasks (one calling the CUBLAS library and the other calling a hand-coded CUDA implementation) and one CPU task (calling the CBLAS library).

The results of two different application versions are presented. In this case, a CPU-only version of matrix multiply was omitted because its performance was too low to be comparable to the other tested versions:

- **GPU-only**: Shown as *GPU* in the charts, only the GPU implementation of the matrix multiply task is given. The task calls the *cublasDgemm* function from CUBLAS library.
- **Heterogeneous**: Shown as *HET* in the charts, corresponds to the heterogeneous application, with three different task implementations: the main implementation is the same as the one given in the *GPU-only* version. The second implementation runs on the GPU and calls a hand-coded CUDA kernel that performs the multiplication. And the last one is an SMP-targeted task for the CPU that calls the CBLAS library to compute the result.

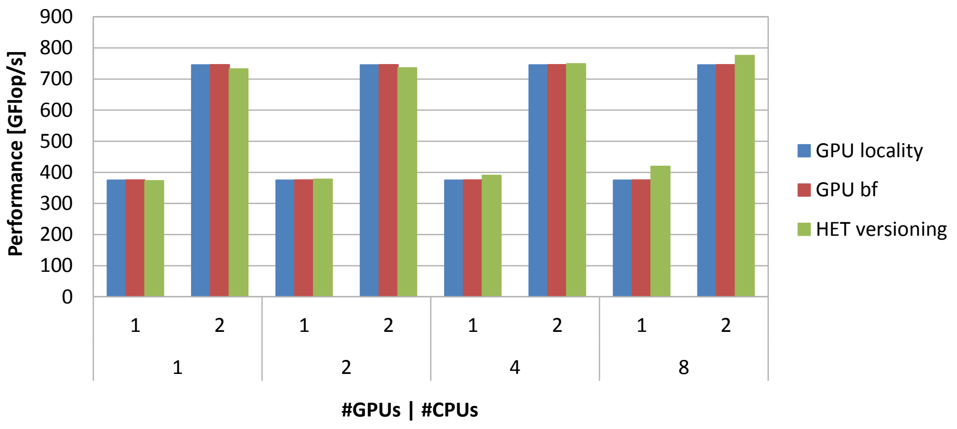


Figure 5.8: Matrix multiply performance results with versioning scheduler

Figure 5.8 shows the performance results of the two tested versions of matrix multiply with the different schedulers. For the *GPU* version, there is no difference between using the locality-aware (*GPU locality*) or the breadth-first scheduler (*GPU bf*) because data locality is well kept in both cases. In addition, the application scales linearly from one to two GPGPUs. There is no difference between using one, two, four or eight CPU threads because there is no parallelism to exploit on the CPU.

The only OmpSs scheduler that exploits the hybrid version of the application is the versioning scheduler, so it can only be executed with this scheduler. The results are represented in the chart as *HET versioning*. For a small number of threads, the overall performance is slightly lower due to several reasons: first, the overheads of sharing data between different memory spaces and second, because the execution time of the CPU version of matrix multiply tasks is much higher than the execution time of the GPU version (CPU task duration is about 60 times the GPU task duration). Nevertheless, the more SMP worker threads collaborate in the application execution, the more benefit versioning scheduler takes despite the fact that more data is transferred.

The performance benefit may look very small in this case, but we cannot expect a huge speed-up because the peak performance of eight CPU cores is still far from the performance of a single GPU: one CPU core represents less than 1% of the machine's peak performance and one GPU represents around 45% of the peak.

Figure 5.9 shows the amount of data transferred for each execution. *GL* represents the *GPU* version run with the locality-aware scheduler, *GB* represents the same version run with the breadth-first scheduler and *HV* represents the *HET* version run with the versioning scheduler. Because part of the computation is done on SMP devices (CPUs) and partial results are shared between CPU and GPU memory spaces, the amount of data transfers for *HET versioning* increases. As the number of SMP workers is increased, memory transfers increase as well, because more work is done by SMP workers and, thus, they need to share more data between CPU and GPU memory spaces. The versioning scheduler is also transferring data between GPGPU devices due to a lack of data locality.

Finally, Figure 5.10 shows the number of times each version is run for the *HET versioning* version. As mentioned before, the application provides three different task versions: *CPU ver-*

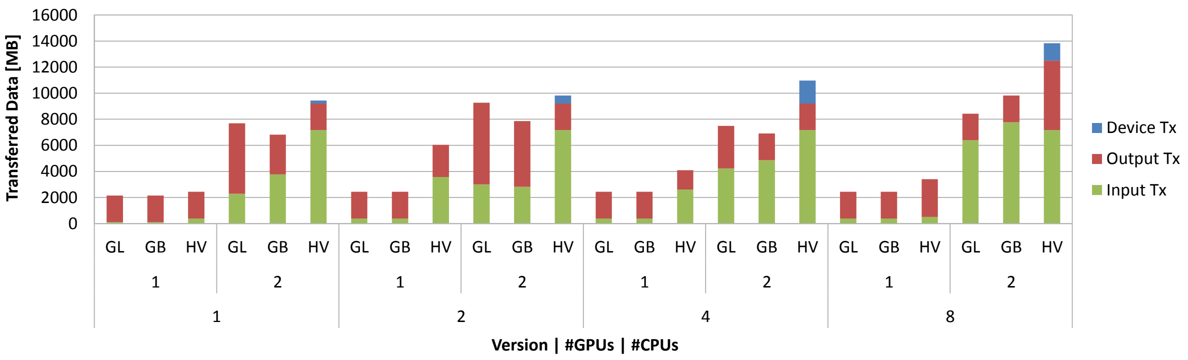


Figure 5.9: Transferred data for matrix multiply with versioning scheduler

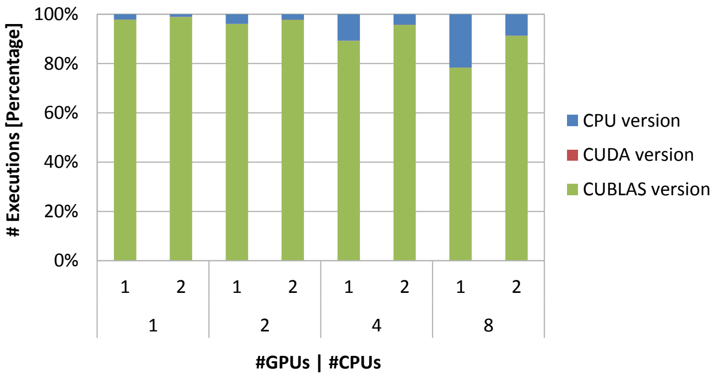


Figure 5.10: Matrix multiply task statistics with versioning scheduler

sion (that calls CBLAS library), *CUDA version* (that calls a hand-coded CUDA kernel) and *CUBLAS version* (that calls CUBLAS library). The fastest implementation (the *CUBLAS version*) is picked most of the times while the *CUDA version* is called only a few times at the beginning of the execution, until versioning realizes that there is a faster implementation for the same device and discards it. The *CUDA version* is represented in the chart in the middle of each bar, but it is almost invisible. The SMP worker threads keep picking the *CPU version* while the GPGPUs are busy (except for the final part of the computation, where only GPGPUs run the fastest implementation to avoid losing performance), and still take about 10% of the work on average that helps improving application's performance. As more SMP workers are added, more work is done by them. And for the same number of SMP worker threads, they do more work when there are less GPGPU resources to do the computation, because the global execution time is longer and they have more time (and more chances) to pick tasks.

Cholesky Factorization. The Cholesky factorization is explained in Section 2.2.2. Single-precision floating-point data was used for the computation. The matrix was organized in blocks of 2048×2048 elements, with a total of 32768×32768 elements. The application had a heterogeneous task for the *spotrf* kernel with two implementations: one CPU version that called CBLAS and one GPU version that called MAGMA. A single GPU task implementation was provided for the other three kernels (*ssyrk*, *sgemm* and *strsm*) that called CUBLAS.

The different application versions used in this test are described below:

- **CPU**: Only the CPU implementation of *spotrf* task is given. However, the other three tasks are always run on the GPU, because running them on the CPU would take too much time for the amount of data they are computing.
- **GPU-only**: Shown as **GPU** in the charts, a single GPU implementation is given for each task. Then, the whole computation runs on the GPGPU.
- **Heterogeneous**: Shown as *HET* in the charts, two different implementations (CPU and GPU versions) are given for the *spotrf* task. The other three tasks are always run on the GPGPU.

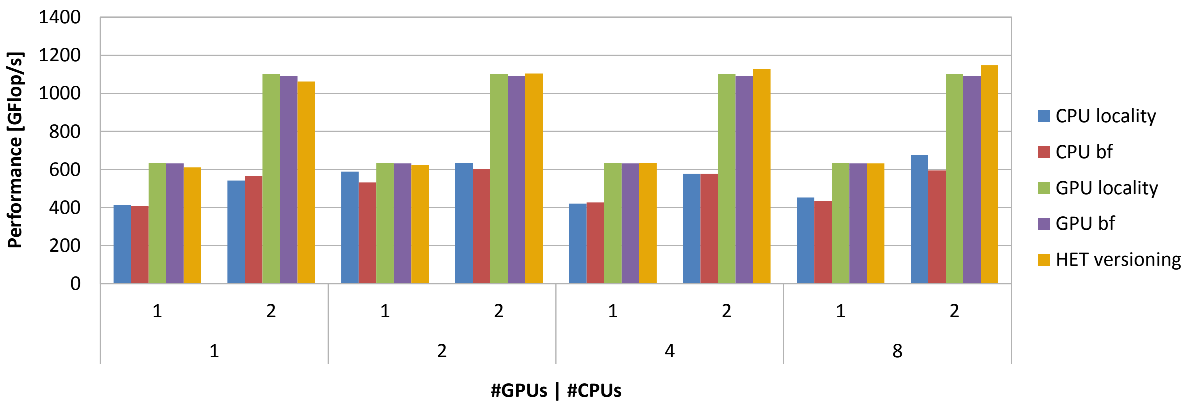


Figure 5.11: Cholesky factorization performance results with versioning scheduler

The performance results of the three Cholesky versions are presented in Figure 5.11. Running the *spotrf* task on the CPU implies several data transfers from and to the GPU memory, plus the CPU version is slower than the GPU version. Thus, the CPU versions (*CPU locality* and *CPU bf*) is the one that gets less performance in all cases.

The *HET versioning* version follows the same performance trend as the matrix multiply test: as the number of SMP workers is increased, the performance of the versioning scheduler gets better than the other tested versions and schedulers. However, the scenario is slightly different in this case: there is a small number of task instances, so the initial learning phase of the

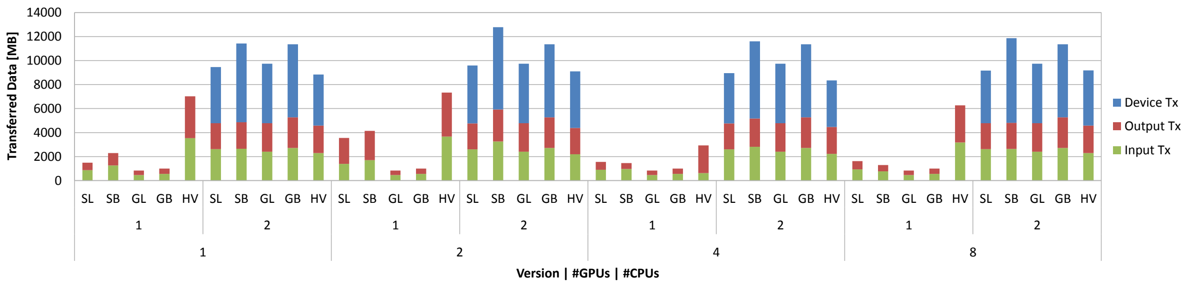


Figure 5.12: Transferred data for Cholesky with versioning scheduler

versioning scheduler impacts on application's performance. However, as shown in Figure 5.12, having more SMP worker threads benefits performance for two reasons: the initial learning phase takes less time and a smaller amount of data is transferred compared to the other schedulers. In this case, tasks share a lot of data, so in order to have good load balancing, data must be continuously exchanged between GPGPUs. Then, the locality-aware scheduler cannot exploit data locality. However, it still gets good performance because data transfers are overlapped with computation.

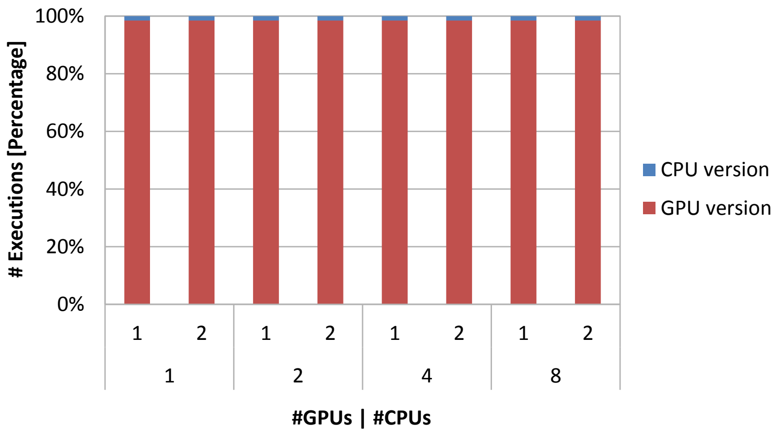


Figure 5.13: Cholesky task statistics with versioning scheduler

Figure 5.13 shows the percentage of times each task version was run for the *HET versioning* version. In contrast with the matrix multiply case and due to Cholesky's data dependency graph complexity, there is not enough parallelism to assign a slow CPU task version to an SMP worker thread. Then, the scheduler decides to assign all the work to the GPGPUs because they are the earliest executors.

PBPI. The general characteristics of this application can be found in Section 2.2.6. The Markov chains used for this test had 50000 elements, using double-precision floating-point data. This application reports its performance as the global execution time, so the performance charts show these results.

As explained before, the three computational loops of the likelihood evaluation were parallelized with three different tasks defined for each iteration of each loop. In order to simplify the presentation of the results, two task implementations were provided for each of the first and second loops. The third loop, although it was annotated as a task, had a single CPU version. Three different versions of the application were evaluated:

- **CPU-only**: Shown as *CPU* in the charts, only the CPU version of each of the three tasks are given. In this case, data always stay in the host memory and no data transfers are needed.

- **GPU**: A single GPU task version is provided for each of the first and second computational loops. The third computational loop has a single CPU task version.

- ❑ **Heterogeneous**: Two implementations are given for each of the first and second loops: the first ones target GPGPUs and the other ones target CPU architectures. The third computational loop has a single CPU task implementation.

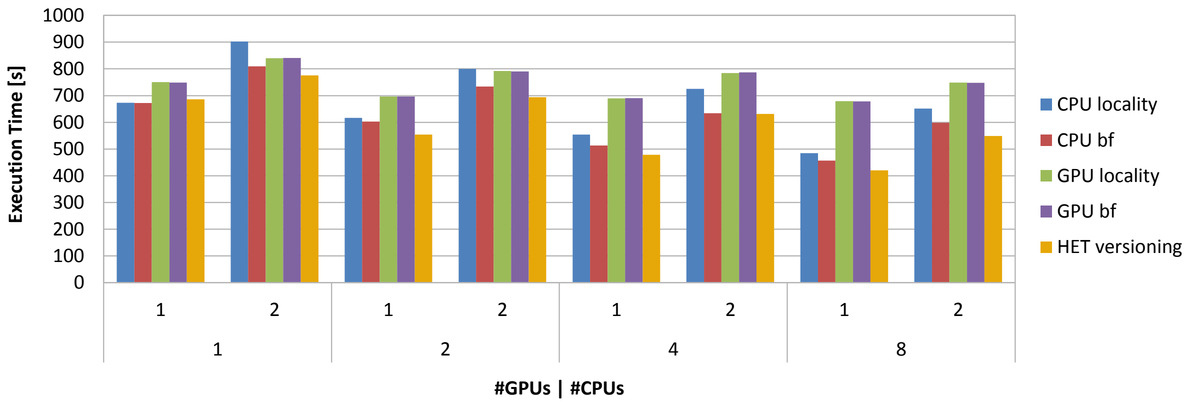


Figure 5.14: PBPI execution time results with versioning scheduler

Figure 5.14 shows the execution time of each version of PBPI. The CPU-only versions (*CPU locality* and *CPU bf*) run faster than the GPU versions (*GPU locality* and *GPU bf*). This is due to the fact that sending the computational work of the first and second loops to the GPGPU is not worthwhile, since a large amount of data needs to be transferred to the GPU memory and transfers cannot be overlapped properly due to data dependencies.

However, the versioning scheduler is able to find the appropriate balance between CPU and GPU execution to take advantage and decrease the execution time. Although the amount of data transfers is higher, as shown in Figure 5.15, it is able to overlap more data transfers with computation thanks to its scheduling decisions. Thus, the global execution time is reduced.

Figures 5.16(a) and 5.16(b) show the percentage of times each task version has been run for the first and second loop respectively. For the first loop, the versioning scheduler decides to send most of the work to the GPGPUs, but the execution of tasks of the second loop is shared between GPGPUs and CPUs. The CPU implementation of the second loop is run in the order of thousands of times, but it may not be clear in the chart, as hundreds of thousands of tasks are run for the second loop. This cooperation helps balancing the trade-off between sending data back and forth from GPU memories and running the tasks on the CPU: the execution

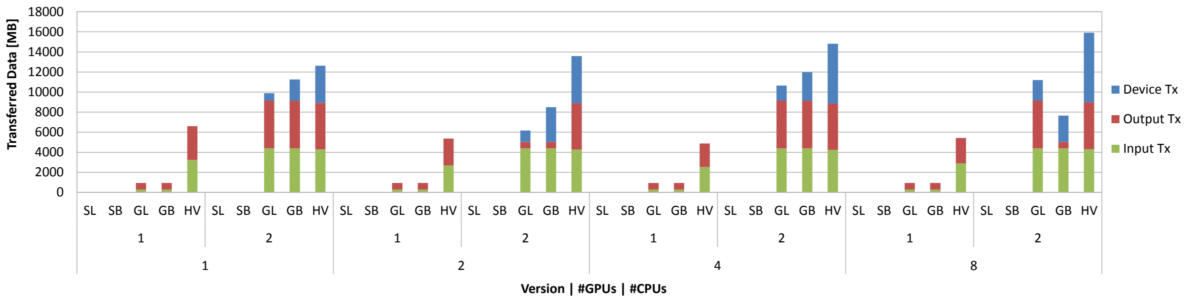
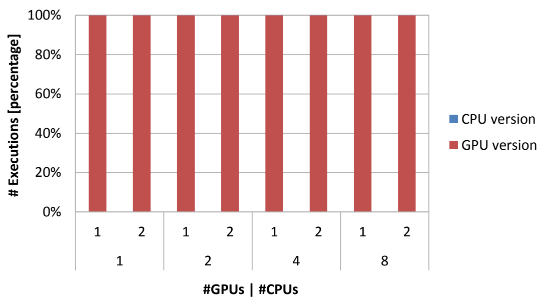
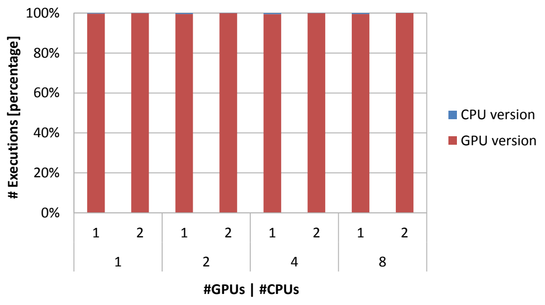


Figure 5.15: Transferred data for PBPI with versioning scheduler



(a) First loop



(b) Second loop

Figure 5.16: PBPI task statistics for first and second loops with versioning scheduler

time of the task is between three and four times slower on the CPU, but the data transfer time is relatively high enough to consider executing all the work on the CPUs.

5.4 SSMART Scheduler

SSMART (Smart Scheduling of Multi-ARchitecture Tasks) is a task scheduler, inspired by versioning, to address the challenges of programming in heterogeneous environments and it has been implemented on top of OmpSs as well. SSMART also supports the concept of *heterogeneous tasks* annotated with the `implements` clause and does not require any additional changes in the user source code with respect to versioning. Moreover, the same Mercurium compiler support as versioning is needed, so it does not require any additional change.

SSMART improves versioning scheduler by tackling its main weaknesses: awareness of data locality, addition of task stealing mechanisms, recognition and propagation of task priorities and profiling of not only heterogeneous tasks, but also regular tasks.

5.4.1 Data Collection

The scheduler reuses the data structures used by versioning, explained in Section 5.3.2 and makes use of another additional table to calculate the global execution time needed to run a task on each processing unit. The results of this table are used to make the scheduling decisions. The bottom part of Figure 5.17 shows an example of this table: for each system processing unit (PU), the following estimated times are used to calculate the estimated global execution time $GblT$ for each task:

- **Estimated busy time (EBT)**: It is an estimation of the time needed by this processing unit to finish the execution of all its assigned tasks.
- **Data transfer time (DtxT)**: It is the time needed to transfer all the task data to the processing unit memory space. This measure only accounts the real data transfers required; so those pieces of data that are already updated and available on the unit's memory space are not counted.
- **Task execution time (TkExT)**: This is the mean time of the previous executions of the task on the processing unit.

Then, the estimated global execution time needed to run a task on a processing unit includes all the aforementioned partial times.

The local task queue of each runtime worker has been modified from the versioning scheduler to support task priorities and task stealing as well.

5.4.2 Runtime Scheduling Policy

The conditions of versioning's initial learning phase have been relaxed and SSMART does not force each task implementation to be run at least λ times. Instead, the initial learning phase has been completely removed and the scheduling decisions are based on the following criteria[6]:

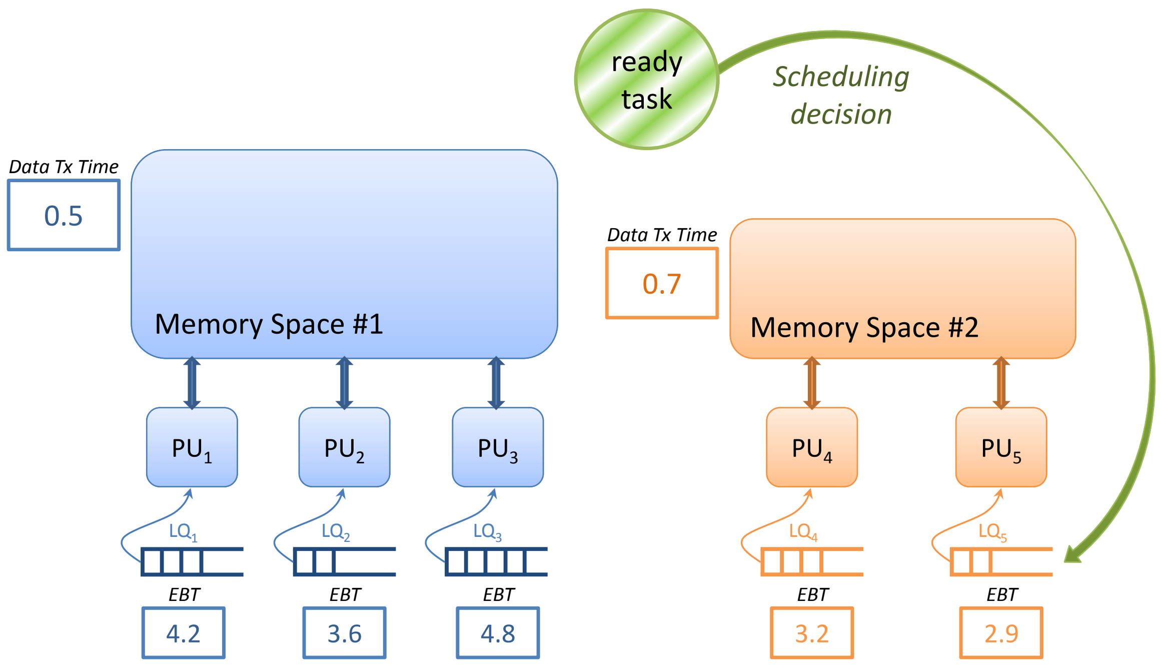
- **Data locality**: For each system unit, SSMART calculates the *data transfer time*, which is the estimated amount of time that would take to transfer all the data that the given task needs (data that is already updated on the unit's memory space is not counted). The time estimation is calculated from memory bandwidth between the involved memory spaces and the amount of data to be transferred.
- **Task execution time**: the scheduler calculates the estimated execution time of the given task on each system unit. Task statistics of previous executions are used to do the calculation. This estimated time is then added to the data transfer time to obtain the total time needed to run the given task on each processing unit.
- **System workload**: Since SSMART schedules tasks in advance, it is able to build a short-term task planning on system units. Each resource has an *estimated busy time* to finish all its assigned work (tasks), so, again, this time is added to the total estimated time for the given task. This results in the *global execution time* of the task and will give an estimation of when the task will complete its execution on each unit.

Then, the task is assigned to the processing unit with the lowest global execution time, or in other words, the unit that will complete its execution at the earliest time. These three criteria allow SSMART to optimally distribute the application execution among all the system processing units. The execution will be balanced, as it is based on the availability of each resource and, in addition, each task will be assigned to the unit that can efficiently run it.

The scheduler supports task priorities as well: this means that ready tasks with higher priorities will be run first and is also able to propagate task priority to a certain number of levels of task predecessors. As explained in Section 5.1, this priority propagation promotes the execution of tasks that lead to higher priority tasks.

When a processing unit (PU) of the system becomes idle, it will ask SSMART for a new task to run. Then, the scheduler's first attempt is to get a ready task from the global ready queue and assign such task to the PU. It may happen that this PU is not the most suitable unit to run the task (it will not complete task execution at the earliest time). Then, such task will be assigned to the most suitable unit and this process is repeated until the PU gets a task.

[6] Note that only those system processing units that can run at least one implementation of the given task are considered.



PU	EBT	DTxT	TkExT	GblT
PU$_1$	4.2	0.5	1.6	**6.3**
PU$_2$	3.6	0.5	1.8	**5.9**
PU$_3$	4.8	0.5	1.8	**7.1**
PU$_4$	3.2	0.7	1.1	**5.0**
PU$_5$	2.9	0.7	1.2	**4.8**

Figure 5.17: SSMART scheduling decisions and timing table

Figure 5.17 shows an example of this process: there are five processing units (PU) in the system with two different memory spaces. Each OmpSs worker thread runs on a different unit. For each worker, SSMART creates a local queue (LQ) to assign tasks to threads and an indicator of the amount of work each thread has (estimated busy time, represented as ETB in the figure). Then, when there is a ready task, the scheduler builds the table shown at the bottom of the figure (described in Section 5.4.1) to find the earliest executor. In this case, PU_5 is the selected worker to run the task. The creation of this table gives SSMART a very powerful feature that could be exploited by other runtime components: the ability to choose the most suitable unit within a subset of all the system units. For example, in NUMA systems, different subsets could be used to prevent CPUs from accessing data of other memory chips. Moreover, it would also be possible to tune the scheduling decisions by giving more weight to certain criteria. For instance, giving more weight to the data transfer time would make SSMART prioritize data locality over the other criteria.

If PU (the processing unit requesting a task) could not get a task after the previous process and there are no more ready tasks, the task stealing process begins. SSMART follows the rules described below, in order. If the attempt to steal a task conforming the first rule fails, then, it will try the other rules successively:

1. Try to steal a task from those processing units that share the same memory space of PU to minimize data transfers. For all the units of the same memory space, it will check which is the busiest unit (the one with the highest estimated busy time. The latest task assigned to this unit will be stolen and assigned to PU[7].

2. The scheduler checks which of the other units of the system are the busiest ones and tries to steal a task from one of such units. In this second attempt, SSMART takes into account the amount of data that will need to transfer and tries to minimize it by not only considering the absolute busiest unit.

A task that can be stolen by another worker is a *stealable task*. There are several restrictions that tasks must accomplish to be stealable tasks, described below:

- A task is stealable only for those worker threads that can execute it. For example, a GPU task is not stealable for an SMP worker thread. Then, heterogeneous tasks are more likely to be stealable by more runtime threads.

- A task that has started its active phase and thus, task input data transfers have started, is not stealable any more.

- A stolen task is not stealable any more. This restriction avoids certain undesired situations. For example, a thread stealing a task from another thread that just had stolen such task.

SSMART is continuously profiling the behavior of the application and recording statistics for both regular and heterogeneous tasks. For each task implementation, its execution time is used to keep the mean execution time of such implementation on a certain unit. More recent values are given a larger weight in the mean computation. Also, its data set size is taken into account, as different data set sizes for the same task implementation will lead to different execution times. This makes the scheduler more flexible and easily adapts to application's behavior, even if it changes over the whole execution.

The following list summarizes the main differences between SSMART and versioning schedulers:

- **Task profiling**: While versioning only takes into account heterogeneous tasks, SSMART profiles and keeps track of both regular and heterogeneous tasks. This allows SSMART to have a better estimation of resource utilization.

- **Data locality**: SSMART takes into account where task data is placed to decide the most suitable unit to run each task. In addition, it calculates, based on memory bandwidth, the estimated time needed to perform the data transfers.

[7] If there are tasks with priorities, higher priority tasks will be stolen first.

- ❏ **Suitability rank**: While versioning only searches for the fastest executor of a task, SSMART, for each pair task-processing unit, builds a suitability rank table to decide which is the best unit to run a task. The suitability rank takes into account three different aspects: task data transfer time, task execution time and unit's estimated busy time.
- ❏ **Task stealing**: SSMART implements a task stealing mechanism that takes into account task data locality and system's workload.
- ❏ **Task priority**: SSMART supports having tasks with different priorities and is able to propagate such priorities to predecessor tasks.

5.4.3 Evaluation

The following sections present the evaluation and performance results of SSMART scheduler tested on a set of OmpSs applications with heterogeneous tasks.

5.4.3.1 Methodology

Three OmpSs applications were selected to run the tests and their performance results are compared with CUDA (GPGPU) and hStreams (Xeon Phi) native versions and other OmpSs schedulers.

Environment. The experiments were run on two different platforms:

- ❏ **Multi-GPU system**: A Linux system with an Intel i7-4820K at 3.7 GHz, 63 GB of memory and four nVIDIA Tesla K40c with 2280 CUDA cores and 12 GB of global memory.
- ❏ **Multi-accelerator system**: A Linux system with an Intel Xeon E5 2x 2680 at 2.6 GHz, 64 GB of memory, an nVIDIA GeForce GTX 480 with 448 CUDA cores and 1.5 GB of global memory and an Intel Xeon Phi 7120P with 61 cores at 1.238 GHz and 16 GB of memory, MPSS version 3.5.

The native CUDA and hStreams codes were compiled with CUDA 6.5 and icc 11.1 respectively. The Intel hStreams library, distributed with MPSS, was used to offload computations to Xeon Phi in both native and OmpSs versions. The OmpSs versions were compiled with OmpSs compiler (using *nvcc* 6.5, *icc* 11.1 and GCC 4.6.4). Optimization level $-O3$ was used in all codes. The same application source code was used for all OmpSs versions. OmpSs runtime version 0.9a with fully-asynchronous (AMA) support for both GPGPUs and Xeon Phi architectures was used.

Experiments. The selected applications were run with different configurations of number of accelerators and data set sizes and analyzed its impact on performance. Results are computed as the mean value of several runs.

For each application, the results of running different native and OmpSs versions are shown: (i) the regular OmpSs application, where each annotated task of the source code is targeting

Configuration	*Application version*	*Scheduler*
OSs-CU-bf	OmpSs CUDA	Breadth-first
OSs-CU-aff	OmpSs CUDA	Affinity-ready
OSs-CU-ssm	OmpSs CUDA	SSMART
OSs-hStr-bf	OmpSs hStreams	Breadth-first
OSs-hStr-aff	OmpSs hStreams	Affinity-ready
OSs-hStr-ssm	OmpSs hStreams	SSMART
OSs-het-bf	OmpSs heterogeneous	Breadth-first
OSs-het-aff	OmpSs heterogeneous	Affinity-ready
OSs-het-ssm	OmpSs heterogeneous	SSMART
CUDA	Native CUDA	–
hStreams	Native hStreams	–

Table 5.2: Description of the different application configurations used

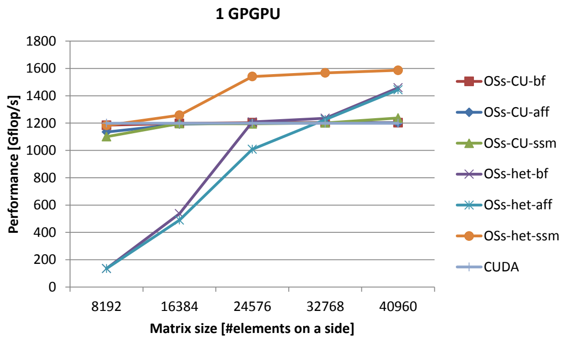
only one device, (ii) the heterogeneous OmpSs version, where all annotated tasks have implementations for all system resources and (iii) the native CUDA and hStreams versions. For the regular OmpSs version of the application, the best combination of task implementations was chosen in order to evaluate the quality of SSMART. Table 5.2 describes the simplified names used in the legends of performance plots. *OmpSs heterogeneous* means that all tasks target all the possible architectures of the system: CPU+GPGPU in the multi-GPU machine and CPU+GPGPU+Xeon Phi in the multi-accelerator machine.

Several schedulers of the OmpSs runtime version used in these experiments support heterogeneous tasks, so the two schedulers that gave the best performance results were used to evaluate the quality of SSMART scheduling:

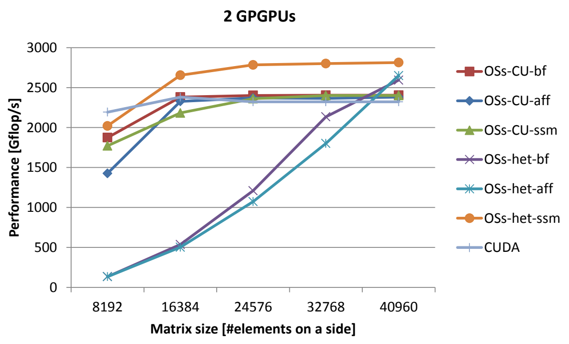
- **Breadth-first scheduler**: the details of this scheduler can be found in Section 2.1.4.3. It basically does a breadth-first task scheduling, but prioritizes the execution of dependency chains. When applications with heterogeneous tasks are run, the scheduler assigns the task to the first idle unit that is requesting work.
- **Affinity-ready scheduler**: this scheduler is explained in Section 5.2 as a contribution of this thesis. In the case of having heterogeneous tasks, the scheduling decisions do not change, as it will still consider all the memory spaces where there is a unit that can run at least one of the task implementations.
- **SSMART scheduler**: The scheduler presented in this section as a thesis contribution.

5.4.3.2 Results

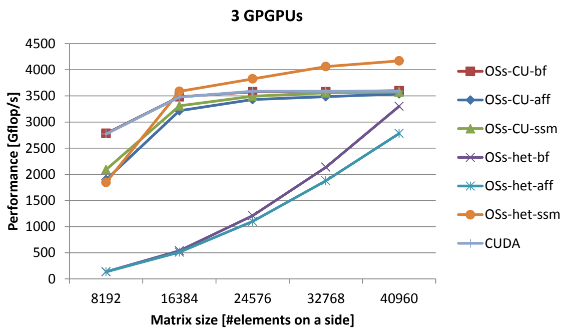
This section presents and compares the results of three applications: matrix multiply, Cholesky factorization and FFT1D run with the three different scheduling policies mentioned before. The evaluation analyzes how the number of resources and application data set size impact on performance.



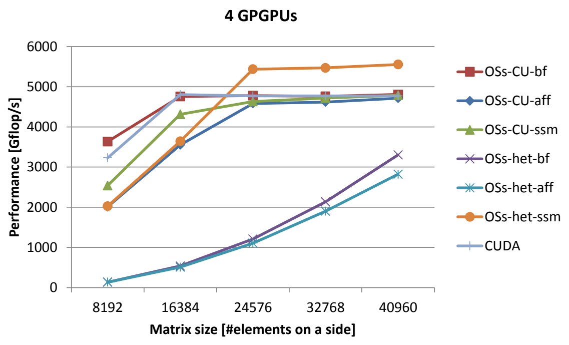
(a) 1 GPGPU



(b) 2 GPGPUs



(c) 3 GPGPUs



(d) 4 GPGPUs

Figure 5.18: Matrix multiply performance results of SSMART on a multi-GPU system

Matrix Multiply. The general details of this application are described in Section 2.2.9. In this test, the GPU computation was done by calling the *cublasDgemm()* function from CUBLAS library. In the native CUDA version, matrices A and C were split into as many chunks as GPGPUs, so each GPGPU received a set of consecutive rows. Matrix B was fully copied to all GPGPU devices. Then, all GPGPUs could compute in parallel with the others. The Xeon Phi computation used the native MKL *dgemm()* implementation, called through the hStreams library. In the native hStreams version, the sample matrix multiply code distributed with hStreams 3.5 (from MPSS 3.5) was used. The host MKL library was used for the CPU computations. In the OmpSs version, each matrix was divided into square blocks of 2048×2048. All versions used double-precision floating-point elements. All available CPUs of each system were used in the case of OmpSs heterogeneous versions.

Figure 5.18 shows the performance results of the tested configurations of matrix multiply on the multi-GPU system. The OmpSs CUDA versions (*OCUbf*, *OCUaf* and *OCUsm*) get approximately the same performance as the native CUDA version, so this means that the overhead introduced by SSMART scheduler is small, especially for larger matrix sizes. However, the OmpSs heterogeneous versions get bad performance for the breadth-first (*OHTbf*) and affinity-ready (*OHTaf*) schedulers: since they are not recording the behavior of each task implementation, there is a considerable load imbalance in task execution. In contrast, the OmpSs heterogeneous version gets the highest performance values with SSMART (*OHTsm*), as CPUs and GPGPUs are efficiently contributing to execute the application. Only for small matrices a small slow-down is observed when using more GPGPUs because there is a lack of

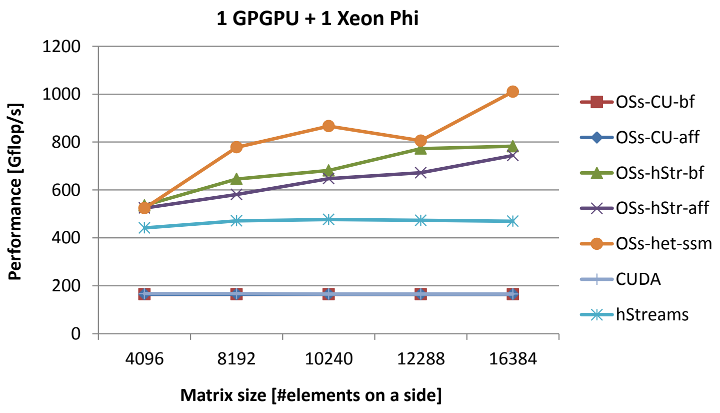


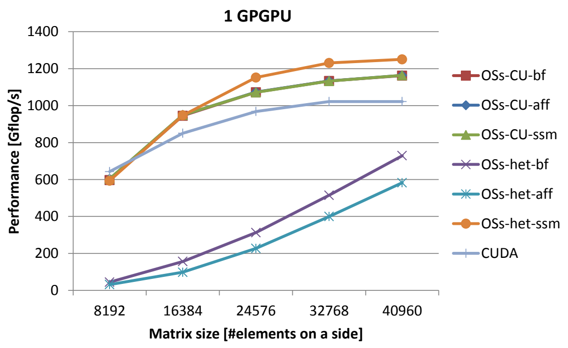
Figure 5.19: Matrix multiply performance results of SSMART on a multi-accelerator system

parallelism. Moreover, the application scales almost linearly as more GPGPUs are used.

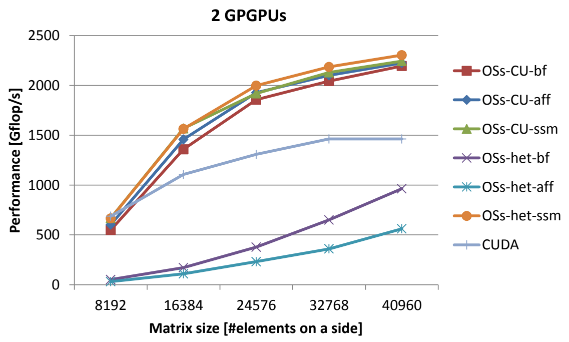
Figure 5.19 shows the performance results of the tested configurations on the multi-accelerator system. In this case, to reduce the complexity of the plot, only the relevant configurations are shown. Even though the GPGPU of this system is less powerful, the same behavior for the OmpSs CUDA versions can be observed: the performance is the same as the CUDA native version. In the case of the hStreams versions, the OmpSs versions get slightly better performance than the native hStreams versions. Finally, the OmpSs SSMART version is able to combine the power of the CPUs, GPGPU and Xeon Phi resources and distribute the work among them all. This again results in the best performance values on this system for matrix multiply.

Cholesky Factorization. The Cholesky factorization is explained in Section 2.2.2. Double-precision floating-point data was used for the computation. The matrix was organized in blocks of 2048×2048 elements. The GPU computations used a customized implementation of *dpotrf* based on its corresponding function from MAGMA library and CUBLAS library was called for the other kernels. The Xeon Phi computation used the corresponding native MKL functions, called through the hStreams library. The host MKL library was used for the CPU computations. The CUDA native version used an OpenMP-like fork-join approach due to its complexity. In the native hStreams version, the sample Cholesky code distributed with hStreams library was used. In the OmpSs version, each kernel was annotated as a task and task data dependencies were managed by the OmpSs runtime. We used different task priorities to give more priority to tasks in the critical path. All available CPUs of the system were used in the case of OmpSs heterogeneous versions and all tasks were heterogeneous, providing one implementation for each available architecture in the system.

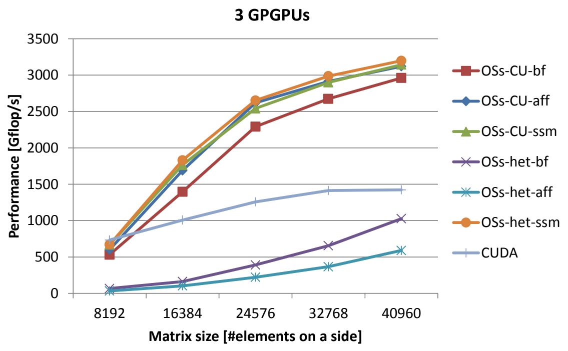
Figure 5.20 shows the performance results for the tested Cholesky configurations in the multi-GPU environment. The CUDA native version is not able to scale across several GPGPUs due to the overhead of the fork-join approach. In contrast, the OmpSs versions are able to achieve higher performance as we increase the number of GPUs. Nevertheless, this application is not able to scale linearly due to the complexity of its task data dependencies. Like in the matrix multiply case, OmpSs heterogeneous versions with breadth-first and affinity-ready schedulers are not able to scale due to a wrong distribution of computations among the available re-



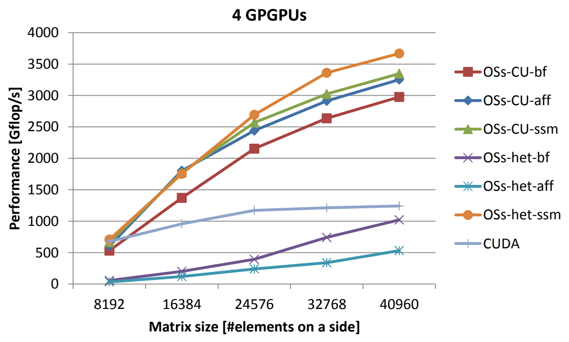
(a) 1 GPGPU



(b) 2 GPGPUs



(c) 3 GPGPUs



(d) 4 GPGPUs

Figure 5.20: Cholesky factorization performance results of SSMART on a multi-GPU system

sources. Regarding the OmpSs CUDA versions, those schedulers that take data affinity into account (affinity-ready and SSMART) perform better than the others (breadth-first), as for this application keeping data locality reduces the number of data transfers significantly. In this case, the application cannot get a great performance improvement by using CPUs and GPGPUs at the same time, as there is a trade-off between the number of resources and the amount of data that needs to be shared among these resources. Thus, the SSMART scheduler slightly enhances Cholesky's performance in the OmpSs heterogeneous version.

Figure 5.21 shows the performance results of the tested configurations on the multi-accelerator system. Like in the matrix multiply plot, only the most relevant versions are shown. The CUDA native version gets low performance because this GPU is less powerful than the ones used in the multi-GPU system. Nevertheless, the OmpSs CUDA versions get fairly good performance. In contrast, the hStreams native implementation is highly optimized for Xeon Phi architectures and the OmpSs versions can hardly get the same performance. The OmpSs heterogeneous version run with the SSMART scheduler has some overhead for smaller matrix sizes, but as the size increases, it gets better performance than the other OmpSs schedulers. Moreover, SSMART even outperforms the native hStreams implementation for the largest data set size because it efficiently uses all the available resources in the system: CPUs, GPGPU and Xeon Phi.

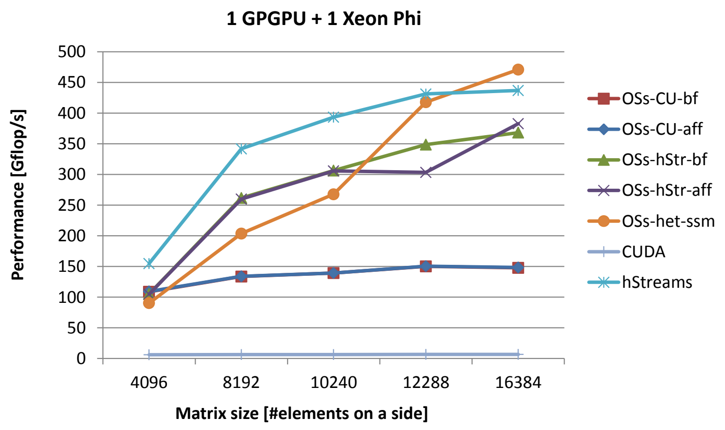
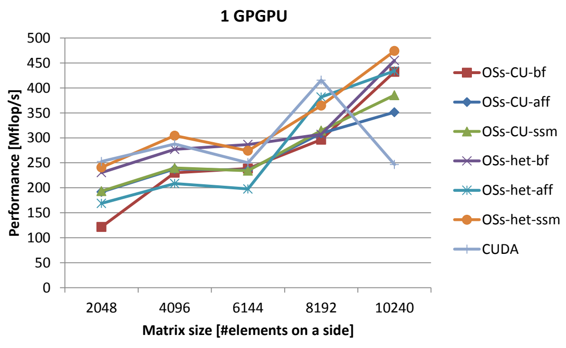


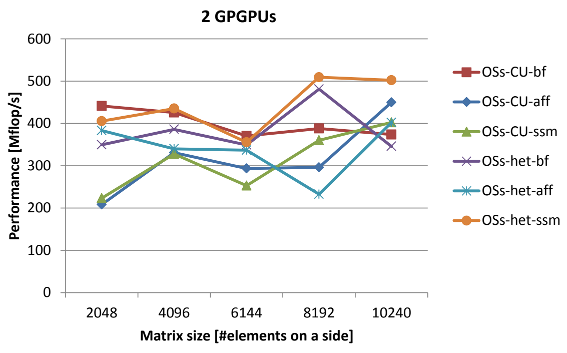
Figure 5.21: Cholesky factorization performance results of SSMART on a multi-accelerator system

FFT1D. The general characteristics of this application can be found in Section 2.2.3. The GPU computation was done by calling the functions from CUFFT library. The native CUDA version was an adapted version based on the *convolutionFFT2D* sample distributed with CUDA 6.5 SDK. Unfortunately, we did not have a multi-GPU implementation of this version due to its complexity. The host MKL library was used for the CPU computations. In the OmpSs versions, each of the steps was translated into several tasks, as explained in the general description. All available CPUs of the system were used in the case of OmpSs heterogeneous versions and all tasks were heterogeneous, providing one implementation for each available architecture in the system.

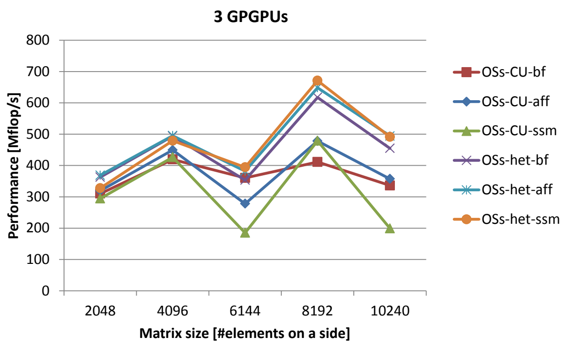
Figure 5.22 shows the performance results of FFT1D. This application is memory-bound and it cannot scale linearly across several GPGPUs. This also introduces a more irregular behavior of the application. However, the OmpSs versions show fair scalability. Several optimizations are used for power-of-two matrix sizes and this is clearly reflected in the performance of 6144 and 10240 matrix sizes. The most interesting fact to highlight in this case is that the OmpSs CUDA version with SSMART scheduler has less performance than the other schedulers because there are more types of tasks to profile and this increases the overhead of this scheduler. However, when the OmpSs heterogeneous version is used, it gets the best performance in almost all cases.



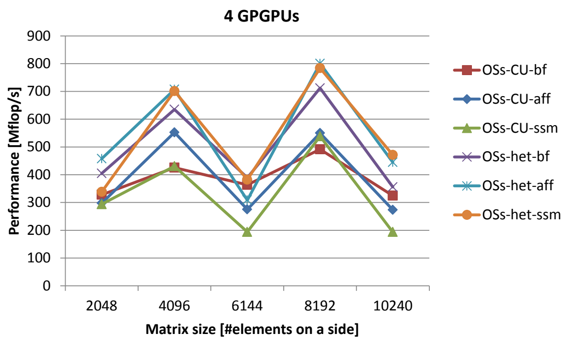
(a) 1 GPGPU



(b) 2 GPGPUs



(c) 3 GPGPUs



(d) 4 GPGPUs

Figure 5.22: FFT1D performance results of SSMART on a multi-GPU system

Chapter 6

Conclusions and Future Work

With the emergence of heterogeneous systems that clearly beat the performance of traditional environments, the programming wall has been hit. The diversity of these new systems presents new programming challenges: on the one hand, each emerging architecture presents its own programming model and, on the other, such programming models excessively expose hardware characteristics to the programmer. This means that, first, programmers need to adapt their codes every time they want to run their applications on a different platform. And second, that their codes must be redesigned to fit the specific characteristics of the particular hardware.

Moreover, programmability is not the only challenge that must be addressed in heterogeneous architectures; planning a good scheduling of applications on these platforms is complicated and machine-dependent. Even though heterogeneous platforms can greatly speed-up applications, making a bad scheduling decision can harm the performance of the whole execution.

This thesis has contributed in both the **programmability** and **work distribution** of applications on heterogeneous systems. This work has proposed several approaches to manage accelerator systems and scheduling techniques with the property that can be applied to runtime frameworks. This property demonstrates that the inherent difficulties of programming heterogeneous systems can be hidden from the programmer side, so that application codes remain clean and clear from all this complexity. The next sections present and discuss the conclusions obtained from these contributions and propose some future work as well.

6.1 Programmability and Accelerator Support

Regarding the programmability aspect of heterogeneous systems, this thesis has proposed two approaches to deal with multi-accelerator systems: a **semi-asynchronous** and a **fully-asynchronous** approaches. These approaches are designed to target different types of accelerators where pieces of code are asynchronously offloaded to such devices and data movements can also be issued asynchronously from the host point of view.

Their objective is to efficiently manage the execution of applications on multi-accelerator systems while hiding to the programmer all the complexity involved in accelerator management, data movements and host-device synchronization.

The semi-asynchronous approach makes use of several accelerator streams to overlap data transfers with computations. Due to hardware restrictions, explicit, blocking synchronization points are introduced to coordinate accelerators and host threads.

As the hardware capabilities of the first accelerators evolved, the semi-asynchronous approach was transformed into the fully-asynchronous design, called AMA (Asynchronous Management of Accelerators). AMA removes the blocking states of host threads devoted to device management by offering instead a blocking-free host-device communication mechanism. For that purpose, accelerator events and callbacks are used to manage communications, computations and data movements.

These designs have been implemented on top of OmpSs, a task-based programming model and framework, to prove their validity. As demonstrated by the experimental results on GPGPU and Intel Xeon Phi platforms, these approaches are able to efficiently offload tasks on such accelerators and transparently manage their data transfers as well. The AMA design has the additional advantage of hiding framework's overheads and devoting the host idle time to do other useful work in the runtime. As a result, our experiments showed that the AMA implementations for GPGPU and Intel Xeon Phi can get up to 2x performance speed-up with respect the semi-asynchronous approach, and in some cases, they even get better performance than their corresponding native, hand-tuned applications.

This natural evolution of both the hardware and software designs demonstrates two facts: first, that programmers cannot rely on hardware capabilities, as they evolve and change relatively fast. This means that *application performance is not guaranteed across hardware generations* and programmers are forced to adapt their source codes to fit new generation hardware characteristics. The second fact is that, actually, *all these changes to fit new hardware capabilities can be done inside runtime frameworks*, so programmers can focus on the development of their applications without having to worry about the particular characteristics of the specific architecture they want to target.

6.1.1 Future Work

As future work, it sounds reasonable to extend the AMA implementation to other asynchronous devices supported by OmpSs, like OpenCL or FPGA devices. Then, the specific implementations may need to be tuned to fit the particular characteristics of such devices.

In addition, in a more experimental area, it would be interesting to consider how the AMA design could fit in a cluster environment. Given the similarities between accelerators and cluster nodes (separate memory spaces, asynchronous operations and communications, etc.), AMA could be redesigned to be even more general by not only targeting accelerators, but also cluster environments.

6.2 Work Distribution and Scheduling

This thesis has contributed in the work distribution aspect by providing several scheduling techniques carefully designed for both accelerators and heterogeneous architectures.

Two scheduling mechanisms have been proposed to target accelerators in task-based environments: the **task priority propagation mechanism** and the **affinity-ready scheduler**.

Given a task-based application with its task data dependency graph, it is important to schedule correctly those tasks in the critical path in order to get good performance. In addition, it is also convenient to schedule first those tasks that open more parallelism. Since the development environment, the OmpSs framework, offers the possibility to assign priorities to tasks, programmers can assign higher priority to those tasks that belong to the critical path or open more parallelism. The proposed priority propagation mechanism *favours the execution of those tasks leading to high priority tasks.* This mechanism is able to propagate the priority of successor tasks up to several levels of parent tasks. This process can be combined with other scheduling policies to help them find the right paths to high priority tasks.

With the objective of *minimizing the amount of data transfers between memory spaces*, the affinity-ready scheduler has been presented. For each task, the scheduler computes the affinity score to each system's memory space. The affinity score counts the amount of data that should be transferred to run the task on such memory space. Then, the task is run on a processing unit with direct access to the memory space that needs less amount of data to be transferred. In addition, several mechanisms are provided to solve affinity score ties and enable task stealing for a better workload balancing.

These two scheduling mechanisms take advantage of the AMA design to use the host spare time to make their scheduling decisions. Thus, they can be used to improve application performance while their implementation does not add runtime overhead. Our performance results showed the importance of making good scheduling decisions to maximize application performance.

In the context of heterogeneous architectures, two scheduling policies have been presented to address the challenges of splitting application execution among all the processing units of such systems: the **versioning** and the **SSMART** schedulers.

They have been implemented on top of the OmpSs task-based programming model and have extended this framework with a new feature: the ability to run heterogeneous tasks (i.e. tasks that provide several implementations, targeting different architectures). They are able to distribute and balance task execution among all system processing units by taking into account resource availability, data transfer costs (only SSMART) and task execution time on each unit. The schedulers record statistics for both heterogeneous tasks and regular tasks (only SSMART) since the application starts and keep updating them for the whole execution so that they can easily adapt if the behavior of the application changes.

With these new schedulers, the programmer can write heterogeneous applications where multiple task implementations targeting multiple devices (CPU, GPGPU, Xeon Phi, etc.) are given. This feature enhances the programmability of applications and makes their maintenance easier, because the programmer, at any time, can develop a new implementation for an already existing task in the code that targets the same or a different device and that can potentially improve application's performance.

The performance results showed that, in most cases, the versioning scheduler outperforms the other existent schedulers for the OmpSs runtime and at the same time, gives more flexibility to the programmer. Only in a few cases the versioning scheduler slightly slows down the application compared to the other OmpSs schedulers.

Even these versioning experimental results were promising, some weaknesses were detected and thus corrected with the SSMART (Smart Scheduling of Multi-ARchitecture Tasks) scheduler. The contributions of SSMART over the versioning scheduler are: (i) *enhanced task profiling*, by not only profiling heterogeneous tasks (like versioning does), but also regular tasks; (ii) SSMART is aware of *data locality* and data transfer times are taken into account to make scheduling decisions; (iii) SSMART creates a *suitability rank* for each pair of task-processing unit, as opposed to versioning, which only searches for the task's fastest executor; (iv) *task stealing* is supported to better balance the execution across system processing units; and (v) *task priority* support is included, as well as the ability to propagate priority to several levels of parent tasks.

From our results in a CPU+multi-GPU and a CPU+GPGPU+Xeon Phi environments, we observed that, in general, the SSMART scheduler reached higher performance than the other tested versions, including hand-tuned native versions for GPGPU and Xeon Phi architectures. Moreover, we have proved that the overhead of the proposed scheduler is negligible for most cases.

This research in scheduling techniques for heterogeneous environments leads us to conclude that *resource cooperation* is usually the best option to efficiently run applications on such platforms and maximize their performance. We believe that each part of the application should be executed on the processing unit that runs it faster. However, *data locality* and *high-priority tasks* play an important role in task scheduling as well. Furthermore, this thesis contributions demonstrate that the burden of task scheduling on accelerators and heterogeneous systems can successfully be implemented inside runtime frameworks and only a minimal help is required from the programmer point of view (e.g. expressing task priorities).

6.2.1 Future Work

In versioning and SSMART, each heterogeneous task has its own execution information depending on its data size. It is true that the execution time of a task can potentially depend on the size of data that it computes or processes. Nevertheless, this implementation decision means that if the data needed by two calls to the same task varies from only one byte, both schedulers will consider that these calls are completely different. Thus, they will not reuse the information collected at the first execution when the task is run for the second time. In this case, it would be better to group data sizes in a reasonable range so that different calls to a task that process different amounts of data of the same magnitude would be grouped together.

Additionally, new features could be added to versioning and SSMART, for example, offering the possibility to receive external hints for task implementations (e.g. better suitability on certain processing units): this information could be given as an XML file, either written by the user, or written by the schedulers themselves from a previous application's execution.

Bibliography

[1] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[2] "How GPU came to be used for general computation," http://igoro.com/archive/how-gpu-came-to-be-used-for-general-computation.

[3] G. Chrysos, "Intel Xeon Phi Coprocessor - the Architecture," *Intel Whitepaper*, 2014.

[4] K. Underwood, "Fpgas vs. cpus: trends in peak floating-point performance," in *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, ser. FPGA '04. New York, NY, USA: ACM, 2004, pp. 171–180. [Online]. Available: http://doi.acm.org/10.1145/968280.968305

[5] "GeForce GTX 980 Ti Specifications," http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-980-ti/specifications.

[6] "GeForce 900 Series," https://en.wikipedia.org/wiki/GeForce_900_series.

[7] "AMD FirePro S9150 Server GPU," https://www.amd.com/Documents/firepro-s9150-datasheet.pdf.

[8] "Intel Xeon Phi Coprocessor Peak Theoretical Maximums," http://www.intel.com/content/www/us/en/benchmarks/server/xeon-phi/xeon-phi-theoretical-maximums.html.

[9] "Intel Xeon Processor E7-8870," http://ark.intel.com/products/53580/Intel-Xeon-Processor-E7-8870-30M-Cache-2_40-GHz-6_40-GTs-Intel-QPI.

[10] "Intel Microprocessor Export Compliance Metrics. Intel Xeon Processor E7-8800 Product Family," http://download.intel.com/support/processors/xeon/sb/xeon_E7-8800.pdf.

[11] "TOP500 Supercomputing Site. June 2015," http://www.top500.org/lists/2015/06.

[12] *CUDA C Programming Guide Version 7.0*, nVIDIA Corporation, March 2015.

[13] Khronos OpenCL Working Group, *The OpenCL Specification, version 2.0*, https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf, 2014.

[14] *Basic Linear Algebra Subprograms*, http://www.netlib.org/blas.

[15] *Linear Algebra PACKage*, http://www.netlib.org/lapack.

[16] *Reference Manual for Intel Math Kernel Library 11.1 Update 4*, Intel Corporation, 2014.

[17] *CUBLAS Library v.7.0*, nVIDIA Corporation, March 2015.

[18] I. C. L. Team, *Matrix Algebra for GPU and Multicore Architectures User Guide v.1.6.2*, University of Tennessee, May 2015.

[19] Intel Corporation, "Running Intel MKL on an Intel Xeon Phi Coprocessor in Native Mode," in *Intel Math Kernel Library for Linux* OS User's Guide 11.1*, 2014.

[20] G. Bansal, C. J. Newburn, P. Besl, R. Deodhar, and R. Narayanaswamy, "Fast matrix computations on asynchronous streams," in *High Performance Parallelism Pearls*, J. Reinders and J. Jeffers, Eds. Morgan Kaufman, 2015.

[21] J. Bueno-Hedo, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, and J. Labarta, "Productive Programming of GPU Clusters with OmpSs," in *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS 2012, May 2012.

[22] J. Planas, R. Badia, E. Ayguade, and J. Labarta, "Self-Adaptive OmpSs Tasks in Heterogeneous Environments," in *IEEE 27th International Parallel & Distributed Processing Symposium (IPDPS)*, 2013, pp. 138–149.

[23] J. Planas, R. M. Badia, E. Ayguade, and J. Labarta, "AMA: Asynchronous Management of Accelerators for Task-based Programming Models," *International Conference on Computational Science (ICCS)*, vol. 51, pp. 130–139, June 2015.

[24] E. Ayguade, R. Badia, P. Bellens, D. Cabrera, A. Duran, M. Gonzalez, F. Igual, D. Jimenez-Gonzalez, J. Labarta, L. Martinell, X. Martorell, R. Mayo, J. Perez, J. Planas, and E. Quintana-Ortí, "Extending OpenMP to Survive the Heterogeneous Multi-core Era," *International Journal of Parallel Programming*, vol. 38, no. 5-6, pp. 440–459, June 2010.

[25] R. Ferrer, J. Planas, P. Bellens, A. Duran, M. Gonzalez, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta, "Optimizing the Exploitation of Multicore Processors and GPUs with OpenMP and OpenCL," in *Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing*, ser. LCPC'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 215–229.

[26] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[27] E. Ayguadé, R. M. Badia, P. Bellens, J. Bueno-Hedo, A. Duran, Y. Etsion, M. Farreras, R. Ferrer, J. Labarta, V. Marjanovic, L. Martinell, X. Martorell, J. M. Pérez, J. Planas, A. Ramirez, X. Teruel, I. Tsalouchidou, and M. Valero, "Hybrid/Heterogeneous Programming with OmpSs and its Software/Hardware Implications," in *Programming Multi-Core*

and Many-Core Computing Systems (Wiley Series on Parallel and Distributed Computing). John Wiley & Sons, Inc., January 2012.

[28] OpenMP ARB, "OpenMP Application Program Interface, v. 4.0," July 2013.

[29] J. Perez, R. Badia, and J. Labarta, "A Dependency-aware Task-based Programming Environment for Multi-core Architectures," *IEEE International Conference on Cluster Computing*, pp. 142–151, 2008.

[30] A. Duran, J. M. Pérez, E. Ayguadé, R. M. Badia, and J. Labarta, "Extending the OpenMP Tasking Model to Allow Dependent Tasks," in *OpenMP in a New Era of Parallelism.* Springer Berlin / Heidelberg, 2008, pp. 111–122.

[31] A. Filgueras, E. Gil, D. Jimenez-Gonzalez, C. Alvarez, X. Martorell, J. Langer, J. Noguera, and K. Vissers, "OmpSs@Zynq All-programmable SoC Ecosystem," in *Proceedings of ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA'14, 2014, pp. 137–146.

[32] L. Martinell, ""Memory usage improvements for the SMPSs runtime"," Master's thesis, Computer Architecture Department, Universitat Politècnica de Catalunya, 2010.

[33] J. M. Perez, R. M. Badia, and J. Labarta, "Handling task dependencies under strided and aliased references," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. New York, NY, USA: ACM, 2010, pp. 263–274.

[34] J. Bueno, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta, "Implementing OmpSs Support for Regions of Data in Architectures with Multiple Address Spaces," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS'13.* New York, NY, USA: ACM, 2013, pp. 359–368.

[35] E. Ayguade, R. M. Badia, D. Cabrera, A. Duran, M. Gonzalez, F. Igual, D. Jimenez, J. Labarta, X. Martorell, R. Mayo, J. M. Perez, and E. S. Quintana-Orti, "A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures," in *IWOMP: Evolving OpenMP in an Age of Extreme Parallelism*, vol. 5568. Germany: Springer, June 2009, pp. 154–167.

[36] NANOS Group, "The Nanos++ runtime library at NANOS group site," https://pm.bsc.es/nanox.

[37] R. Ferrer, J. Planas, P. Bellens, A. Duran, M. Gonzalez, X. Martorell, R. Badia., E. Ayguade, and J. Labarta, "Optimizing the Exploitation of Multicore Processors and GPUs with OpenMP and OpenCL," in *Proceedings of the 23rd International Workshop on Languages and Compilers for Parallel Computing (LCPC2010)*, October 2010.

[38] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta, " Productive Cluster Programming with OmpSs ," in *Euro-Par 2011 Parallel Processing, Bordeaux, France, August, 2011, Proceedings, Part I*, ser. Lecture Notes in Computer Science, E. Jeannot, R. Namyst, and J. Roman, Eds., vol. 6852. Springer, 2011, pp. 555–566.

[39] MPI Forum, "MPI: A Message Passing Interface Standard, v 3.1," June 2015.

[40] A. Rico, A. Duran, F. Cabarcas, A. Ramirez, Y. Etsion, and M. Valero, “Trace-driven Simulation of Multithreaded Applications,” in *2011 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2011, pp. 87–96.

[41] NANOS Group, “The Mercurium compiler at NANOS group site,” https://pm.bsc.es/mcxx.

[42] M. H. Lars Nyland and J. Prins, “Chapter 31: Fast N-Body Simulation with CUDA,” in *GPU Gems 3*, 1st ed., H. Nguyen, Ed. Addison-Wesley Professional, 2007.

[43] J. Dongarra and P. Luszczek, *HPC Challenge: Design, History, and Implementation Highlights*. CRC Computational Science Series, 2013.

[44] OpenMP ARB, “OpenMP Application Program Interface, v. 3.0,” May 2008.

[45] L. A. Smith, “Mixed mode MPI/OpenMP programming,” *UK High-End Computing Technology Report*, 2000.

[46] Intel Corporation, “Intel Cilk Plus,” in *User and Reference Guide for the Intel C++ Compiler 15.0*, 2014.

[47] Cray Inc., “Chapel Language Specification, v. 0.97,” April 2015.

[48] T. Knight, J. Park, M. Ren, M. Houston, M. Erez, K. Fatahalian, A. Aiken, W. Dally, and P. Hanrahan, “Compilation for Explicitly Managed Memory Hierarchies,” in *Proc. of the 2007 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2007.

[49] M. Bauer, J. P. Clark, E. Schkufza, and A. Aiken, “A CUDA Runtime Target for the Sequoia Compiler,” in *GTC 2010 Research Posters*. GPU Technology Conference, 2010.

[50] R. Landaverde, T. Zhang, A. Coskun, and M. Herbordt, “An investigation of Unified Memory Access performance in CUDA,” in *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, Sept 2014, pp. 1–6.

[51] M. Fatica, “Accelerating linpack with cuda on heterogeneous clusters,” in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2. New York, NY, USA: ACM, 2009, pp. 46–51.

[52] T. Hamada and K. Nitadori, “190 tflops astrophysical n-body simulation on a cluster of gpus,” in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–9.

[53] N. Karunadasa and D. D. N. Ranasinghe, “Accelerating high performance applications with CUDA and MPI,” in *Proceedings of the Fourth International Conference on Industrial and Information Systems (ICIIS 2009)*, Sri Lanka, 28-31 December 2009.

[54] O. S. Lawlor, “Message passing for GPGPU clusters: CudaMPI,” in *Proceedings of the IEEE International Conference on Cluster Computing*, 2009, pp. 1–8.

[55] S. J. Pennycook, S. D. Hammond, S. A. Jarvis, and G. R. Mudalige, "Performance analysis of a hybrid mpi/cuda implementation of the naslu benchmark," *SIGMETRICS Performance Evaluation Review*, vol. 38, pp. 23–29, March 2011.

[56] S. Pennycook, S. Hammond, S. Wright, J. Herdman, I. Miller, and S. Jarvis, "An investigation of the performance portability of opencl," *Journal of Parallel and Distributed Computing*, vol. 73, no. 11, pp. 1439 – 1450, 2013, novel architectures for high-performance computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731512001669

[57] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi, "Evaluating Performance and Portability of OpenCL Programs," in *Proc. Automatic Performance Tuning*, 2010.

[58] *HSA Platform System Architecture Specification, v 1.0*, HSA Foundation, January 2015.

[59] *HSA Programmer's Reference Manual: HSAIL Virtual ISA and Programming Model, Compiler Writer, and Object Format (BRIG), v 1.0*, HSA Foundation, March 2015.

[60] *HSA Runtime Programmer's Reference Manual, v 1.0*, HSA Foundation, February 2015.

[61] OpenACC standard, *The OpenACC Application Programming Interface, v. 2.0*, August 2013.

[62] Intel Corporation, "Intel-Specific Pragma Reference," in *User and Reference Guide for the Intel C++ Compiler 15.0*, 2014.

[63] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation : Practice & Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[64] C. Augonnet and R. Namyst, "A unified runtime system for heterogeneous multicore architectures," in *Proceedings of the International Euro-Par Workshops 2008, HPPC'08*, ser. Lecture Notes in Computer Science, vol. 5415. Las Palmas de Gran Canaria, Spain: Springer, Aug. 2008, pp. 174–183.

[65] C. Johns and D. Brokenshire, "Introduction to the Cell Broadband Engine Architecture," *IBM Journal of Research and Development*, vol. 51, pp. 503–519, September 2007.

[66] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst, "Data-Aware Task Scheduling on Multi-accelerator Based Platforms," in *IEEE 16th International Conference on Parallel and Distributed Systems (ICPADS)*, 2010, pp. 291–298.

[67] A.-E. Hugo, A. Guermouche, P.-A. Wacrenier, and R. Namyst, "Composing multiple starpu applications over heterogeneous machines: A supervised approach," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, May 2013, pp. 1050–1059.

[68] "StarPU website," http://starpu.gforge.inria.fr.

[69] P. Cooper, U. Dolinsky, A. F. Donaldson, A. Richards, C. Riley, and G. Russell, "Offload – Automating Code Migration to Heterogeneous Multicore Systems," in *Lecture Notes in Computer Science, HiPEAC Conference 2010*, 2010, pp. 307–321.

[70] J. Pienaar *et al.*, "MDR: performance model driven runtime for heterogeneous parallel platforms," in *ICS'11*, 2011, pp. 225–234.

[71] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *MICRO 42*, 2009, pp. 45–55.

[72] T. D. Han and T. S. Abdelrahman, "hiCUDA: High-Level GPGPU Programming," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, pp. 78–90, 2011.

[73] R. Dolbeau, S. Bihan, and F. Bodin, "HMPP: A Hybrid Multi-core Parallel Programming Environment," in *Workshop on General Processing Using GPUs*, 2007.

[74] OpenHMPP Consortium, "OpenHMPP Concepts and Directives," https://en.wikipedia.org/wiki/OpenHMPP.

[75] M. Pharr and W. Mark, "ispc: A spmd compiler for high-performance cpu programming," in *Innovative Parallel Computing (InPar), 2012*, May 2012, pp. 1–13.

[76] S. Lee and R. Eigenmann, "OpenMPC: Extended OpenMP Programming and Tuning for GPUs," in *Proc. of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. USA: IEEE Computer Society, 2010, pp. 1–11.

[77] S. Ueng, M. Lathara, S. Baghsorkhi, and W. Hwu, "CUDA-lite: Reducing GPU Programming Complexity," in *21st Languages and Compilers for Parallel Computing (LCPC)*, 2008.

[78] U. Consortium, "UPC Language Specifications v1.2," May 2005.

[79] L. Chen, L. Liu, S. Tang, L. Huang, Z. Jing, S. Xu, D. Zhang, and B. Shou, "Unified parallel C for GPU clusters: Language extensions and compiler implementation," in *Languages and Compilers for Parallel Computing.* Springer, 2011, pp. 151–165.

[80] Mark Harris, *How to Overlap Data Transfers in CUDA C/C++*, http://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc, nVIDIA Parallel Forall Blog, 2012.

A

Nanos++ Runtime Options

A.1 Nanos++ GPU Runtime Options

Table A.1 summarizes all the GPU-related options that can be configured at run time to run OmpSs applications. The table shows both the configuration option name (that should be used inside the *NX_ARGS* environment variable) and its corresponding environment variable (which can be used standalone without the need for *NX_ARGS*). The accepted and default values are also shown along with a brief description.

Configuration option/ Environment variable	**Accepted values**	**Default value**	**Description**
--enable-cuda NX_ENABLECUDA	yes/no	Enabled	Enable or disable the use of GPUs with CUDA
--gpus NX_GPUS	integer	All GPUs	Defines the maximum number of GPUs to use
--gpu-warmup NX_GPUWARMUP	yes/no	Enabled	Enable or disable warming up the GPU before running user's code
--gpu-prefetch NX_GPUPREFETCH	integer	1	Defines the maximum number of tasks to prefetch
--gpu-concurrent-exec NX_GPU_CONCURRENT_EXEC	yes/no	Enabled	Enable or disable concurrent kernel execution, if supported by the hardware
--gpu-overlap NX_GPUOVERLAP	yes/no	Disabled	Set whether GPU computation should be overlapped with all data transfers, whenever possible, or not
--gpu-overlap NX_GPUOVERLAP	yes/no	Enabled	Set whether GPU computation should be overlapped with all data transfers, whenever possible, or not. *Disabling this option is discouraged and should only be done for debugging purposes*
--gpu-max-memory NX_GPUMAXMEM	integer	No limit	Defines the maximum amount of GPU memory (in bytes) to use for each GPU. If this number is below 100, the amount of memory is taken as a percentage of the total device memory
--gpu-pinned-buffers NX_GPU_PINNED_BUFFERS	yes/no	Enabled	Set whether GPU component should allocate pinned buffers used by data transfers
--gpu-cache-policy NX_GPU_CACHE_POLICY	wt/wb/ nocache	wb	Defines the cache policy for GPU devices: write-through, write-back or do not use cache
--gpu-cublas-init NX_GPUCUBLASINIT	yes/no	Disabled	Enable or disable CUBLAS initialization

Table A.1: OmpSs GPU-related configuration options

A.2 Nanos++ Xeon Phi Runtime Options

Table A.2 summarizes all the Xeon Phi-related options that can be configured at run time to run OmpSs applications. The table shows both the configuration option name and its corresponding environment variable. The accepted and default values are also shown along with a brief description.

Configuration option/ Environment variable	**Accepted values**	**Default value**	**Description**
--enable-hstreams NX_ENABLE_HSTREAMS	yes/no	Disabled	Enable or disable the use of Xeon Phi cards with hStreams
--num-hstr-devs NX_HSTR_DEVS	integer	All Xeon Phis	Defines the maximum number of Xeon Phi cards to use
--hstr-partitions NX_HSTR_PARTITIONS	integer	4	Defines the number of partitions per device
--hstr-warmup NX_HSTR_WARMUP	yes/no	Enabled	Enable or disable warming up the Xeon Phi card before running user's code
--hstr-prefetch NX_HSTR_PREFETCH	integer	1	Defines the maximum number of tasks to prefetch
--hstr-wait-in-tx NX_HSTR_WAIT_IN_TX	yes/no	Disabled	Enable or disable waiting for input transfers before launching the kernel
--hstr-max-memory NX_HSTR_MAXMEM	integer	No limit	Defines the maximum amount of device memory (in bytes) to use for each card. If this number is below 100, the amount of memory is taken as a percentage of the total device memory
--hstr-cache-policy NX_HSTR_CACHE_POLICY	wt/wb/ nocache	wb	Defines the cache policy for Xeon Phi devices: write-through, write-back or do not use cache
--verbose-hstreams NX_VERBOSE_HSTREAMS	yes/no	Disabled	Enable or disable the hStreams library verbose mode

Table A.2: OmpSs Xeon Phi-related configuration options

B

Application Source Code Comparison

This section presents a comparison between different implementations of the tiled matrix multiply code first described in Section 2.1.6.

Table B.1 compares the OmpSs version for GPGPUs with a C sequential version for a CPU and a native CUDA version for a GPGPU. The table spreads across several pages.

Description / Code in OmpSs GPGPU	**Sequential C code (CPU)**	**Native CUDA code (CUDA)**

// Runtime initialization and allocation of additional support variables		`cudaStream_t * streams = (cudaStream_t *) malloc(M*N * sizeof(cudaStream_t));`
		`cudaEvent_t * events = (cudaEvent_t *) malloc(M*N*L * sizeof(cudaEvent_t));`
		`cublasHandle_t handle;`
		`cudaSetDevice(0);`
		
		`for (i = 0; i < M*N; i++)`
		`   cudaStreamCreate(&streams[i]);`
		
		`for (i = 0; i < M*N*L; i++)`
		`   cudaEventCreate(&events[i]);`
		
		`cublasCreate(&handle);`

Description / Code in OmpSs GPGPU	Sequential C code (CPU)	Native CUDA code (CUDA)

// Data allocation
double **a = malloc(M*L *
sizeof(double *));
double **b = malloc(L*N *
sizeof(double *));
double **c = malloc(M*N *
sizeof(double *));
 	
for (i = 0; i < M*L; i++) {
a[i] = nanos_malloc_pinned_cuda(
BS*BS * sizeof(double));
}
 	
for (i = 0; i < L*N; i++) {
b[i] = nanos_malloc_pinned_cuda(
BS*BS * sizeof(double));
}
 	
for (i = 0; i < M*N; i++) {
c[i] = nanos_malloc_pinned_cuda(
BS*BS * sizeof(double));
}
	double **a = malloc(M*L *
	sizeof(double *));
	double **b = malloc(L*N *
	sizeof(double *));
	double **c = malloc(M*N *
	sizeof(double *));
	
	for (i = 0; i < M*L; i++) {
	a[i] = malloc(BS*BS *
	sizeof(double));
	}
	
	for (i = 0; i < L*N; i++) {
	b[i] = malloc(BS*BS *
	sizeof(double));
	}
	
	for (i = 0; i < M*N; i++) {
	c[i] = malloc(BS*BS *
	sizeof(double));
	}
		double **a = malloc(M*L *
		sizeof(double *));
		double **b = malloc(L*N *
		sizeof(double *));
		double **c = malloc(M*N *
		sizeof(double *));
		double **Ad = malloc(M*L *
		sizeof(double *));
		double **Bd = malloc(L*N *
		sizeof(double *));
		double **Cd = malloc(M*N *
		sizeof(double *));
		
		for (i = 0; i < M*L; i++) {
		cudaMallocHost(&a[i], BS*BS *
		sizeof(double));
		cudaMalloc(&Ad[i], BS*BS *
		sizeof(double));
		}
		
		for (i = 0; i < L*N; i++) {
		cudaMallocHost(&b[i], BS*BS *
		sizeof(double));
		cudaMalloc(&Bd[i], BS*BS *
		sizeof(double));
		}
		
		for (i = 0; i < M*N; i++) {
		cudaMallocHost(&c[i], BS*BS *
		sizeof(double));
		cudaMalloc(&Cd[i], BS*BS *
		sizeof(double));
		}

// Data transfers to device		for (i = 0; i < M*L; i++) cudaMemcpyAsync(&Ad[i], &a[i], BS*BS * sizeof(double), cudaMemcpyHostToDevice, streams[i%(M*N)]); for (i = 0; i < L*N; i++) cudaMemcpyAsync(&Bd[i], &b[i], BS*BS * sizeof(double), cudaMemcpyHostToDevice, streams[i%(M*N)]); for (i = 0; i < M*N; i++) cudaMemcpyAsync(&Cd[i], &c[i], BS*BS * sizeof(double), cudaMemcpyHostToDevice, streams[i]); cudaDeviceSynchronize();

Description / Code in OmpSs GPGPU	Sequential C code (CPU)	Native CUDA code (CUDA)

// Kernel computation
#pragma omp target device (cuda) copy_deps
#pragma omp task inout([NB*NB]C) in([NB*NB]A, [NB*NB]B)
void dgemm_tile (double *A, double *B, double *C, int NB) {
  double alpha = 1.0;
  cublasSetStream(nanos_get_cublas_handle(), nanos_get_kernel_execution_stream ()) ;
  cublasDgemm_v2(nanos_get_cublas_handle(), CUBLAS_OP_T, CUBLAS_OP_T, NB, NB, NB, &alpha, A, NB, B, NB, &alpha, C, BS);
}
 	
for (i = 0;i < M; i++) {
  for (j = 0; j < N; j++) {
    for (k = 0; k < L; k++) {
      dgemm_tile(a[i*L+k], b[k*N+j], c[i*N+j], BS);
    }
  }
}
	for (i = 0;i < M; i++) {
	  for (j = 0; j < N; j++) {
	    for (k = 0; k < L; k++) {
	      cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, BS, BS, BS, 1.0, a[i*L+k], BS, b[k*N+j], BS, 1.0, c[i*N+j], BS);
	    }
	  }
	}
		double alpha = 1.0;
		
		for (i = 0;i < M; i++) {
		  for (j = 0; j < N; j++) {
		    for (k = 0; k < L; k++) {
		      cublasSetStream(handle, streams[i*M+j]);
		      cublasDgemm_v2(handle, CUBLAS_OP_T, CUBLAS_OP_T, BS, BS, BS, &alpha, Ad[i*L+k], BS, Bd[k*N+j], BS, &alpha, Cd[i*N+j], BS);
		      cudaEventRecord(events[(i*M+j) * N + k]);
		    }
		  }
		}

// Synchronization
#pragma omp taskwait
		// Option 1: Device
		cudaDeviceSynchronize();
		
		// Option 2: Streams
		for (j = 0; j < M * N; j++)
		  cudaStreamSynchronize(streams[j]);
		
		// Option 3: Events
		for (j = 0; j < M * N * L; j++)
		  cudaEventSynchronize(events[j]);

// Data transfers from device
*// The **taskwait** directive already transfers data back to host*
		for (i = 0; i < M*N; i++)
		  cudaMemcpyAsync(&c[i], &Cd[i], BS*BS * sizeof(double), cudaMemcpyDeviceToHost, streams[i]);
		
		cudaDeviceSynchronize();

Description / Code in OmpSs GPGPU	Sequential C code (CPU)	Native CUDA code (CUDA)

// Data deallocation
for (i = 0; i < M*L; i++) {
  nanos_free_pinned_cuda(a[i]);
}
 	
for (i = 0; i < L*N; i++) {
  nanos_free_ pinned_cuda (b[i]);
}
 	
for (i = 0; i < M*N; i++) {
  nanos_free_ pinned_cuda (c[i]);
}
 	
free(a);
free(b);
free(c);
	for (i = 0; i < M*L; i++) {
	  free(a[i]);
	}
	
	for (i = 0; i < L*N; i++) {
	  free(b[i]);
	}
	
	for (i = 0; i < M*N; i++) {
	  free(c[i]);
	}
	
	free(a);
	free(b);
	free(c);
		for (i = 0; i < M*L; i++) {
		  cudaFreeHost(a[i]);
		  cudaFree(Ad[i]);
		}
		
		for (i = 0; i < L*N; i++) {
		  cudaFreeHost(b[i]);
		  cudaFree(Bd[i]);
		}
		
		for (i = 0; i < M*N; i++) {
		  cudaFreeHost(c[i]);
		  cudaFree(Cd[i]);
		}
		
		free(a);
		free(b);
		free(c);
		free(Ad);
		free(Bd);
		free(Cd);

// Runtime finalization and deallocation of additional support variables		cublasDestroy(handle);
		
		for (i = 0; i < M*L*N; i++)
		  cudaEventDestroy(events[i]);
		
		for (i = 0; i < M*N; i++)
		  cudaStreamDestroy(streams[i]);
		
		free(events);
		free(streams);

Table B.1: Source code comparison between OmpSs, sequential C and native CUDA

Table B.2 compares the OmpSs version with the OpenMP and the native hStreams versions that offload the computation to a Xeon Phi coprocessor. The table spreads across several pages.

Description / Code in OmpSs Xeon Phi	OpenMP 4.0 (Xeon Phi)	Native hStreams code (Xeon Phi)

// Runtime initialization and allocation of additional support variables		hStreams_app_init(4, M*N);
		HSTR_EVENT * events =
		    (HSTR_EVENT *) malloc(M*N*L *
		    sizeof(HSTR_EVENT));

// Data allocation
double **a = malloc(M*L *
    sizeof(double *));
double **b = malloc(L*N *
    sizeof(double *));
double **c = malloc(M*N *
    sizeof(double *));
 	
for (i = 0; i < M*L; i++) {
  a[i] = nanos_malloc_hstreams(
    BS*BS * sizeof(double));
}
 	
for (i = 0; i < L*N; i++) {
  b[i] = nanos_malloc_hstreams(
    BS*BS * sizeof(double));
}
 	
for (i = 0; i < M*N; i++) {
  c[i] = nanos_malloc_hstreams(
    BS*BS * sizeof(double));
}
	double **a = malloc(M*L *
	    sizeof(double *));
	double **b = malloc(L*N *
	    sizeof(double *));
	double **c = malloc(M*N *
	    sizeof(double *));
	
	for (i = 0; i < M*L; i++) {
	  a[i] = malloc(BS*BS *
	    sizeof(double));
	}
	
	for (i = 0; i < L*N; i++) {
	  b[i] = malloc(BS*BS *
	    sizeof(double));
	}
	
	for (i = 0; i < M*N; i++) {
	  c[i] = malloc(BS*BS *
	    sizeof(double));
	}
		double **a = malloc(M*L *
		    sizeof(double *));
		double **b = malloc(L*N *
		    sizeof(double *));
		double **c = malloc(M*N *
		    sizeof(double *));
		
		for (i = 0; i < M*L; i++) {
		  a[i] = (double *) malloc(BS*BS *
		    sizeof(double));
		  hStreams_app_create_buf(a[i],
		    BS*BS*sizeof(double));
		}
		for (i = 0; i < L*N; i++) {
		  b[i] = (double *) malloc(BS*BS *
		    sizeof(double));
		  hStreams_app_create_buf(b[i],
		    BS*BS*sizeof(double));
		}
		
		for (i = 0; i < M*N; i++) {
		  c[i] = (double *) malloc(BS*BS *
		    sizeof(double));
		  hStreams_app_create_buf(c[i],
		    BS*BS*sizeof(double));
		}

// Data transfers to device		for (i = 0; i < M*L; i++)
		  hStreams_app_xfer_memory(a[i],
		    a[i], BS*BS * sizeof(double), i,
		    HSTR_SRC_TO_SINK, NULL);
		
		for (i = 0; i < L*N; i++)
		  hStreams_app_xfer_memory(b[i],
		    b[i], BS*BS * sizeof(double), i,
		    HSTR_SRC_TO_SINK, NULL);
		
		for (i = 0; i < M*N; i++)
		  hStreams_app_xfer_memory(c[i],
		    c[i], BS*BS * sizeof(double), i,
		    HSTR_SRC_TO_SINK, NULL);
		
		hStreams_app_thread_sync();

Description / Code in OmpSs Xeon Phi	OpenMP 4.0 (Xeon Phi)	Native hStreams code (Xeon Phi)
// Kernel computation #pragma omp target device (hstreams) copy_deps #pragma omp task inout([NB*NB]C) in([NB*NB]A, [NB*NB]B) void dgemm_tile (double *A, double *B, double *C, int NB) { cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, NB, NB, NB, 1.0, A, NB, B, NB, 1.0, C, BS); } for (i = 0;i < M; i++) { for (j = 0; j < N; j++) { for (k = 0; k < L; k++) { dgemm_tile(a[i*L+k], b[k*N+j], c[i*N+j], BS); } } }	for (i = 0;i < M; i++) { for (j = 0; j < N; j++) { for (k = 0; k < L; k++) { #pragma omp target device(0) map(to:a[0:BS*BS],b[0:BS*BS]) map(tofrom:c[0:BS*BS]) cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, BS, BS, BS, 1.0, a[i*L+k], BS, b[k*N+j], BS, 1.0, c[i*N+j], BS); } } }	for (i = 0;i < M; i++) { for (j = 0; j < N; j++) { for (k = 0; k < L; k++) { hStreams_app_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, BS, BS, BS, 1.0, a[i*L+k], BS, b[k*N+j], BS, 1.0, c[i*N+j], BS, i*M+j, &events[(i*M+j)*N+k]); } } }
// Synchronization #pragma omp taskwait		*// Option 1: Device* hStreams_app_thread_sync(); *// Option 2: Streams* for (j = 0; j < M * N; j++) hStreams_StreamSynchronize(j); *// Option 3: Events* hStreams_app_event_wait(M*N*L, events);
*// Data transfers from device The **taskwait** directive already transfers data back to host*		for (i = 0; i < M*N; i++) hStreams_app_xfer_memory(c[i], c[i], BS*BS * sizeof(double), i, HSTR_SINK_TO_SRC, NULL); hStreams_app_thread_sync();
// Data deallocation for (i = 0; i < M*L; i++) { nanos_free_hstreams(a[i]); } for (i = 0; i < L*N; i++) { nanos_free_hstreams(b[i]); } for (i = 0; i < M*N; i++) { nanos_free_hstreams(c[i]); } free(a); free(b); free(c);	for (i = 0; i < M*L; i++) { free(a[i]); } for (i = 0; i < L*N; i++) { free(b[i]); } for (i = 0; i < M*N; i++) { free(c[i]); } free(a); free(b); free(c);	for (i = 0; i < M*L; i++) { free(a[i]); hStreams_DeAlloc(a[i]); } for (i = 0; i < L*N; i++) { free(b[i]); hStreams_DeAlloc(b[i]); } for (i = 0; i < M*N; i++) { free(c[i]); hStreams_DeAlloc(c[i]); } free(a); free(b); free(c);

Description / Code in OmpSs Xeon Phi	OpenMP 4.0 (Xeon Phi)	Native hStreams code (Xeon Phi)

// Runtime finalization and deallocation of additional support variables		`free(events);`
		`hStreams_app_fini();`

Table B.2: Source code comparison between OmpSs, OpenMP and native hStreams

Table B.3 summarizes the differences between all the codes. Taking the sequential C version as the basis, it shows the number of additional lines, variables and code needed by each version. The last row of the table reflects which codes can run on a multi-device environment.

Concept	OmpSs	OpenMP 4.0	CUDA	hStreams
Total number of additional lines of code	GPGPU: 8 Xeon Phi: 6	1	42	21
Total number of additional support variables	–	–	1 array of M*N elems. (streams), 1 array of M*N*L elems. (events), 1 opaque pointer (CUBLAS), 1 matrix of M*L elems. (dev. addr.), 1 matrix of L*N elems. (dev. addr.), 1 matrix of M*N elems. (dev. addr.)	1 array of M*N*L elems. (events)
Total number of unique API calls used	GPGPU: 4 Xeon Phi: 2	–	16	7
Total number of API calls used	GPGPU: 9 Xeon Phi: 6	–	29	16
Total number of unique compiler directives used	GPGPU: 3 Xeon Phi: 3	1	–	–
Total number of compiler directives used	GPGPU: 3 Xeon Phi: 3	1	–	–
Code can run on multi-device as is	GPGPU: Yes Xeon Phi: Yes	No No	No No	No No

Table B.3: Summary of the source code comparison