Memory Hierarchies for Future HPC Architectures

Víctor García Flores

A Dissertation
Submitted in fulfillment of the requirements
for the degree of Doctor of Philosophy / Doctor per la UPC
to the Department of Computer Architecture
Universitat Politècnica de Catalunya

Advisor: Nacho Navarro
Co-advisors: Antonio J. Peña, Eduard Ayguade

Barcelona
September 2017

Abstract

Efficiently managing the memory subsystem of modern multi/manycore architectures is increasingly becoming a challenge as systems grow in complexity and heterogeneity. From multicore architectures with several levels of on-die cache to heterogeneous systems combining graphics processing units (GPUs) and traditional general-purpose processors, the once simple Von Neumann machines have transformed into complex systems with a high reliance on an efficient memory subsystem. In the field of high performance computing (HPC) in particular, where massively parallel architectures are used and input sets of several terabytes are common, careful management of the memory hierarchy is crucial to exploit the full computing power of these systems.

The goal of this thesis is to provide computer architects with valuable information to guide the design of future systems, and in particular of those more widely used in the field of HPC, *i.e.*, symmetric multicore processors (SMPs) and GPUs. With that aim, we present an analysis of some of the inefficiencies and shortcomings of current memory management techniques and propose two novel schemes leveraging the opportunities that arise from the use of new and emerging programming models and computing paradigms.

The first contribution of this thesis is a block prefetching mechanism for task-based programming models. Using a task-based programming model simplifies parallel programming and allows for better resource utilization in the large-scale supercomputers used in the field of HPC, while enabling sophisticated memory management techniques. The scheme proposed relies on a memory-aware runtime system to guide prefetching while avoiding the main drawbacks of traditional prefetching mechanisms, *i.e.*, cache pollution, thrashing and lack of timeliness. It leverages the information provided by the user about tasks' input and output data to prefetch contiguous blocks of memory that are certain to be useful. The proposed scheme targets SMPs with large cache hierarchies and uses heuristics to dynamically decide the best cache level to prefetch into without evicting useful data.

The focus of this thesis then turns to heterogeneous architectures combining GPUs and traditional multicore processors. The current trend towards tighter coupling of GPU and CPU enables new collaborative computations that tax the memory subsystem in a different manner than previous heterogeneous computations did, and requires careful analysis to understand the trade-offs that are to be expected when designing future memory organizations.

The second contribution is an in-depth analysis on the impact of sharing the last-level cache between GPU and CPU cores on a system where the GPU is integrated on the same die as the CPU. The analysis focuses on the effect that a shared cache can have on collaborative computations where GPU and CPU threads concurrently work on a problem and share data at fine granularities. The results presented here show that sharing the last-level cache is largely beneficial as it allows for better resource utilization. In addition, the experimental evaluation shows that collaborative computations benefit significantly from the faster CPU-GPU communication and higher cache hit rates that a shared cache level provides.

The final contribution of this thesis analyzes the inefficiencies and drawbacks of demand paging as currently implemented in discrete GPUs by NVIDIA. Then, it proposes a novel memory organization and dynamic migration scheme that allows for efficient data sharing between GPU and CPU, specially when executing collaborative computations where data is migrated back and forth between the two separate memories. This scheme migrates data at cache line granularities transparently to the user and operating system, avoiding false sharing and the unnecessary data transfers that occur on the current demand paging mechanism.

The results show that the proposed scheme is able to outperform the baseline system by reducing the migration latency of data that is copied multiple times between the two memories. In addition, analysis of different interconnect latencies shows that fine-grained data sharing between GPU and CPU is feasible as long as future interconnect technologies achieve four to five times lower round-trip times than PCI-Express 3.0.

Contents

List of Figures

List of Tables

Glossary

AMAT	Average Memory Access Time
API	Application Programming Interface
APU	Accelerated Processing Unit
CPU	Central Processing Unit
DDR	Double Data Rate
DLP	Data-Level Parallelism
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
GHB	Global History Buffer
GPU	Graphics Processing Unit
GPGPU	General Purpose Computing on GPU
GDDR	Graphics Double Data Rate
HPC	High-Performance Computing
ILP	Instruction-Level Parallelism
IPC	Instructions Per Cycle
ISA	Instruction Set Architecture
LLC	Last-Level Cache

LLT	Line Location Table
LRU	Least Recently Used
LTE	Location Table Entry
MDTE	Multicore Data Transfer Engine
NoC	Network-on-Chip
NUMA	Non-Uniform Memory Access
OS	Operating System
PCIe	PCI-Express
PTE	Page Table Entry
RMW	Read Modify Write
RPT	Reference Prediction Table
RTT	Round-Trip Time
SDRAM	Synchronous Dynamic Random Access Memory
SIMD	Single-Instruction Multiple-Data
SIMT	Single-Instruction Multiple-Thread
SM	Streaming Multiprocessor
SMP	Symmetric Multicore Processor
SoC	System-on-Chip
SRAM	Static Random Access Memory
SVM	Shared Virtual Memory
TLB	Translation Lookaside Buffer
TLP	Thread-Level Parallelism

UM	Unified Memory
UVA	Unified Virtual Addressing

Chapter 1

Introduction

Riding on the self-fulfilled words of Gordon Moore back in the 1960s, microprocessor design saw a prolonged period of performance improvements during the 1990s. New technology developments and architectural enhancements provided year over year performance gains of ~1.5x [1] for more than a decade. Unfortunately, memory technology improvements during the same period of time were more limited, creating the gap in performance between the processor and off-chip memory shown in Figure 1.1. This gap, known as the Memory Wall [2], kept widening as core clock frequencies increased and as novel micro-architectural improvements allowed processors to further exploit instruction-level parallelism (ILP).

By the early 2000s, power and temperature constraints (the so-called Power Wall [3]) had caused the stagnation of core clock frequencies, and the ILP achievable through micro-architectural improvements was generally believed to be beyond the point of diminishing returns [1]. Processor manufacturers soon realized a paradigm shift was necessary and transitioned to the first commodity SMPs. Those first dual-core processors signified the beginning of the multicore era and the shift from ILP-driven performance gains to thread-level parallelism (TLP)-based speedup.

Today, with an increasing number of cores per chip and multiple hardware threads per core, the Memory Wall is is still very much present. In the field of HPC, in particular, the top positions of the Top500 list of supercomputers [4] are already employing manycore processors with hundreds of hardware threads per node. The memory subsystem of such systems must, therefore, sustain the traffic generated by multiple concurrent threads without sacrificing fairness and within a constrained power envelope.

As a consequence, a wide range of new memory technologies and organizations have emerged to fulfill the requirements of these memory-hungry architectures. From software-managed scratchpads giving the user full control of data movement, to 3D die-stacked memories providing one order of magnitude higher bandwidth than traditional memory technologies, or simply by integrating additional and larger levels of on-chip cache memories,

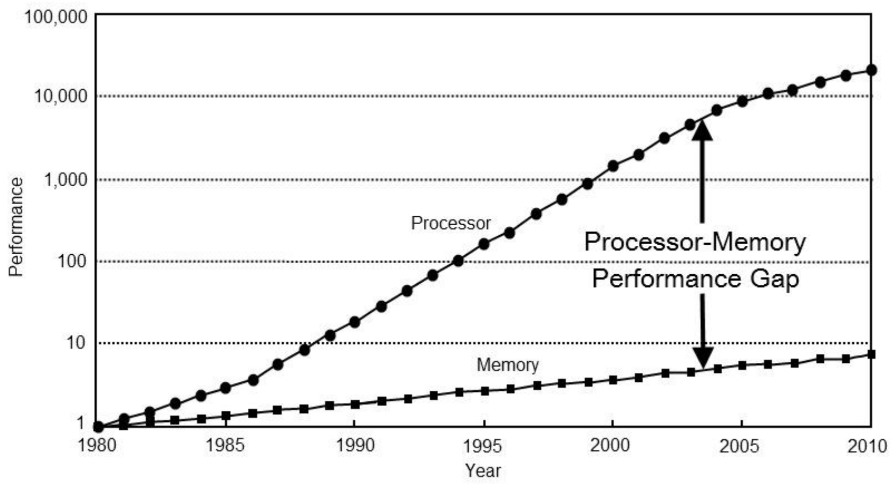


Figure 1.1: Processor-memory performance gap. From ”Computer Architecture: A Quantitative Approach” by John L. Hennessy, David A. Patterson.

the memory hierarchy of current systems is becoming increasingly complex and difficult to manage efficiently.

The cache hierarchy, in particular, is a fundamental part of the memory subsystem in modern architectures. Current processors employ a multi-level hierarchy of on-die cache memories in order to reduce memory access times and bridge the processor-memory performance gap. Static random access memory (SRAM) caches exploit the temporal locality commonly found in CPU applications, providing access times one order of magnitude lower than off-chip dynamic random access memory (DRAM) [1], while reducing the pressure on the interconnect fabric and memory controllers. Their importance is such that the on-chip real-estate devoted to the cache hierarchy may even overshadow that of the computing elements themselves. Utilizing these resources optimally is therefore paramount to achieve the peak performance these architectures are capable of.

One of the techniques extensively used in modern processors to leverage the cache hierarchy is data prefetching. The goal of prefetching is to hide the latency of accessing off-chip memory, avoiding pipeline stalls derived from memory instructions missing in the cache hierarchy. Data prefetching schemes attempt to predict the memory that will be referenced in the future and fetch it in advance, moving it from high-latency DRAM memory into the faster on-die caches. Prefetching can broadly be divided into hardware based or software based. Software prefetching requires executing special prefetch instructions that are typically inserted by the compiler in an optimization pass. Hardware prefetching schemes require specialized hardware, known as prefetch engines, that analyzes the stream of memory accesses attempting to find patterns to predict future memory references.

Nowadays, most processors implement at least one level of hardware prefetching, and in many cases, multiple schemes are implemented for the different levels of the cache hierarchy [5]. Although prefetching schemes have become very efficient at predicting future memory references, their effectiveness depends on the algorithms and data structures used by the application. No single prefetching mechanism has been found that consistently obtains large performance gains in all kinds of applications [6]. Furthermore, even the most sophisticated prefetching techniques may degrade performance by polluting the cache with unnecessary data or simply by prefetching data too early or too late [7, 8]. Accurate prefetching is therefore needed to ensure system performance is not degraded.

In order to exploit the computing power of current multicore architectures and efficiently manage their complex memory organizations, scientists resort to new programming models and runtime systems that can assist the hardware on the task of memory management [9, 10, 11]. These new programming models can ease the tasks of programming parallel algorithms, while their memory-aware runtime systems offer a valuable opportunity for memory optimizations such as data prefetching. Using the information available to the runtime to guide prefetching can avoid the main drawbacks of traditional schemes.

Still, efficiently managing the memory hierarchy is a difficult task as processors become more complex and heterogeneous. The traditional SMP model where several homogeneous cores share a common memory pool is being abandoned for heterogeneous architectures. Multiprocessors with non-uniform memory access times (NUMA) [12], system-on-chip (SoC) with differently sized cores [13] or processors with GPUs integrated on the same die [14] are some examples of the heterogeneity we can find in current systems. This heterogeneity adds another layer of complexity to the task of memory management, as computing elements with very different characteristics must share data and system resources.

In particular, the emergence in the field of HPC of heterogeneous systems composed of commodity multicore processors and GPUs has led to a brand new world of scientific heterogeneous computing. GPUs are massively parallel processors specialized to exploit Data-Level Parallelism (DLP) in a Single-Instruction Multiple-Data (SIMD) fashion (sometimes called Single-Instruction Multiple-Thread, or SIMT). Initially designed for graphics processing, GPUs have found their way into general-purpose computing, and more specifically into the field of HPC, due to their enormous computing capabilities and energy efficiency. As an example of their prevalence in HPC, 34 out of the top 50 supercomputers in the last Green500 list of the most energy-efficient supercomputers use GPUs from NVIDIA or AMD [15].

The large majority of GPUs found today in the market are *discrete* devices connected to a host machine through an expansion bus, *e.g.*, PCI-Express (PCIe) for x86 systems or NVLink for POWER architectures. Discrete GPUs have their own pool of specialized high-bandwidth memory, requiring data to be copied back and forth between host and device. Yet, the trend in the last few years has been towards logical and physical integration of GPU and CPU. We find examples of physical integration in the latest chips from Intel and AMD, which integrate the GPU on-die with the CPU [14, 16]. These architectures forgo the separate address spaces and provide a unified memory pool that can be access directly by GPU and CPU cores.

Logical integration of GPU and CPU is a natural step taken by GPU manufacturers to improve the programmability of their devices and to make general-purpose computing on GPUs (GPGPU) more accessible to the public. Until recently, heterogeneous computing in systems with discrete GPUs required the programmer to explicitly manage the two separate memory pools and copy data back and forth between them. Newer products from NVIDIA improve programmability with features such as a shared virtual address space that allows GPU and CPU to use the same pointers to shared data structures [17], and more recently, automatic data movement between memories performed transparently to the user by the CUDA runtime [18]. Unfortunately, these features are still relatively new and suffer from inefficiencies that cause performance loss compared to using fine-tuned manual data movement.

The trend towards tighter integration and the use of emerging heterogeneous computing frameworks with features such as shared virtual memory and system-wide atomic operations [19] have opened the design space for collaborative computations. On the traditional heterogeneous model the host does little to no computation, usually relegated to copying the data to the GPU and waiting for the results. In collaborative computations, on the other hand, the algorithms are partitioned and each part is assigned to the computing element it is best suited for, *i.e.*, regions with data and/or thread parallelism are assigned to the GPU, while regions with low parallelism are assigned to the larger CPU cores that can exploit higher ILP.

Physical and logical integration of GPU and CPU is therefore desirable as it improves programmability and allows for collaborative computations, but it also presents new challenges that computer architects must consider when designing tightly coupled heterogeneous architectures. Physical integration leads to resource sharing among computing elements with widely different characteristics, such as large out-of-order CPU cores and small in-order GPU cores. Which resources should be shared and how to best manage them to guarantee fairness and maximize performance are still open questions that require detailed analysis.

Logical integration on discrete architectures also leads to new issues that should be explored. The fine-grained data sharing patterns seen when executing collaborative computations greatly differ from that of traditional heterogeneous computations where data is copied in bulk transfers only at kernel boundaries. The memory organization of current heterogeneous architectures is designed for the traditional computational model and is therefore inefficient when collaborative computations are executed.

1.1 Thesis Objectives and Contributions

In this dissertation we analyze the challenges of efficiently managing the memory hierarchy in current multi/manycore processors. In particular, we analyze the architectures more commonly used in the field of HPC, *i.e.*, multicore SMPs and GPUs. Our analysis focuses on the use of new and emerging programming and computational models, and how their use modifies the trade-offs encountered when designing the memory subsystem.

The objective of this work is to understand the shortcomings of current memory management schemes and identify possible design improvements. Our goal is to aid computer architects by providing new techniques that can guide the design decisions of future architectures. In order to meet these goals, this dissertation makes the contributions we describe in the following.

1.1.1 Adaptive Runtime-Assisted Block Prefetching

We identify an opportunity for efficient data management when using a task-based programming model with a memory-aware runtime system. We propose an adaptive software-based prefetching scheme that leverages runtime system memory awareness to avoid the main drawbacks of typical prefetchers, *i.e.*, cache pollution and cache thrashing.

Our scheme leverages the information about the tasks' input and output data to prefetch only data that is certain to be needed, avoiding cache pollution. In addition, knowing in advance the memory region required by each task allows the runtime to generate prefetch instructions for blocks of data instead of one cache line at at time, improving efficiency. The runtime-directed scheme minimizes cache thrashing by dynamically deciding the cache level to prefetch into based on the amount of data calculated to fit without evicting the current working set. Lastly, it leverages the information about the execution path to initiate the prefetch with enough time to ensure, to a certain degree, that data will be ready by the time it is needed.

To support the prefetching scheme, we propose a small DMA-like controller to asynchronously manage prefetching. This simple hardware structure receives prefetch commands generated by the runtime system, performs address translation, and initiates the movement of data from main memory to the cache hierarchy.

1.1.2 Last-Level Cache Sharing on Integrated Heterogeneous Architectures

The second contribution of this dissertation is an in-depth analysis on the effect of sharing the last-level cache on a heterogeneous architecture integrating the GPU on-die with the CPU. We provide an analysis of two memory configurations: a *shared* configuration where GPU and CPU have a common L3 last-level cache they can access equally, and a *split* configuration where each have their own private last-level cache.

In this part of the thesis we focus specifically on the behavior of the memory subsystem when executing collaborative heterogeneous computations. In these applications, GPU and CPU share data at fine granularities during the computation and therefore the design of the memory hierarchy has a significant impact on the performance of the applications. In addition, we also evaluate the two memory organizations with a set of traditional heterogeneous benchmarks where GPU and CPU only share data at kernel boundaries.

This analysis shows the benefits and drawbacks of a shared last-level cache and provides insights in order to design the memory subsystem of future integrated architectures.

1.1.3 Efficient Data Sharing on Heterogeneous Architectures

The final contribution of this work is a memory organization and dynamic data migration scheme for heterogeneous architectures with discrete GPUs. We identify the shortcomings of the current dynamic data management scheme in NVIDIA GPUs and propose a mechanism that efficiently shares data between host and device, especially when executing collaborative computations where data is migrated multiple times between the two memories.

We analyze the inefficiencies of demand paging as it is currently implemented in the latest family of GPUs by NVIDIA, namely, false sharing caused by the large granularity at which data is migrated and unnecessary long-latency page fault handling on every migration. We then propose a memory organization and a dynamic migration scheme that efficiently moves data between the two memories transparently to the user and the operating system (OS).

Our scheme reduces the granularity of data transfers to cache lines from full OS-defined memory pages and avoids paying multiple times the page fault handling of data that is migrated more than once. We leverage the observation that the page table of the heterogeneous process very rarely needs to be modified during runtime, and therefore copying only once each page table entry to the GPU is sufficient to perform virtual address translation. In addition, we provide an analysis of different interconnect latencies to evaluate the feasibility of fine-grained memory transfers.

1.2 Thesis Organization

The rest of this dissertation is organized as follows:

Chapter 2: State of the Art introduces the state of the art and provides some background to understand the rest of the work done in this thesis. This chapter explores previous work on prefetching and similar proposals that leverage a runtime system. It then covers the topic of resource sharing on integrated heterogeneous architectures and collaborative computations. It concludes with state of the art on dynamic data movement schemes for heterogeneous architectures with discrete GPUs.

Chapter 3: Methodology presents the methodology followed throughout the thesis. It introduces the simulation infrastructure used in each chapter, as well as the workloads and metrics used to evaluated the proposed work.

Chapter 4: Adaptive Runtime-Assisted Block Prefetching covers the first contribution, providing motivation, a description of the target architecture, implementation details and the results obtained.

Chapter 5: Last-Level Cache Sharing on Integrated Heterogeneous Architectures covers the second contribution, motivating it and providing details about the two memory organizations evaluated. It then presents the evaluation with an in-depth analysis of the results.

Chapter 6: Efficient Data Sharing on Heterogeneous Architectures covers the last contribution of this thesis. It introduces the motivation behind this technique, details about the target architecture, main implementation details and the evaluation of the proposed scheme.

Chapter 7: Conclusions and Future Work closes this dissertation, reviewing the contributions, summarizing the insights obtained during the thesis and detailing some potential lines of future work.

Chapter 2
State of the Art

This chapter presents the state of the art relevant to this dissertation, introducing the concepts and ideas that provide a background to understand and frame the rest of the work. It first introduces task-based programming models and the concept of data prefetching, discussing relevant prefetching schemes found in the literature and the metrics commonly used to evaluate them. It then focuses on heterogeneous architectures and the programming models used for heterogeneous computing.

2.1 Task-Based Programming Models

Modern multicore processors integrate tens of cores with multiple hardware threads per core. Furthermore, supercomputers are built by aggregating several processors in a node and connecting hundreds of nodes together. Programming applications to take advantage of the enormous computing power of these systems is a complex endeavor and has spurred the development of new programming models.

Task-based programming models, in particular, attempt to simplify parallel programming by introducing the concept of *tasks*, *i.e.*, self-contained portions of code that run serially but can run concurrently with other tasks. A runtime system manages the execution order of the tasks, guaranteeing that data dependencies between tasks are maintained.

Tasks provide an intuitive way for programmers to break down complex algorithms into and to exploit the available parallelism of modern machines. Some examples of task-based programming models include: Cilk [11], OpenMP [9], Sequoia [20], OmpSs [10], StarPU [21], X10 [22], Chapel [23] and Intel TBB [24]

Task-based dataflow programming models, in particular, provide automatic dependency tracking by the runtime system, further simplifying parallel programming. Programmers require only annotating their code with information about the input and output data used by each task [9, 10, 21].

```
#pragma omp task in(a, b) inout(c)
void sgemm_t(float a[M][M], float b[M][M], float c[M][M]);

#pragma omp task inout(a)
void spotrf_t(float a[M][M]);

#pragma omp task in(a) inout(b)
void strsm_t(float a[M][M], float b[M][M]);

#pragma omp task in(a) inout(b)
void ssyrk_t(float a[M][M], float b[M][M]);

---------------------------------------------

float A[N][N][M][M]; // NxN blocked matrix, with MxM blocks
for (int j = 0; j<N; j++) {
   for (int k = 0; k<j; k++)
      for (int i = j+1; i<N; i++)
         sgemm_t(A[i][k], A[j][k], A[i][j]);

   for (int i = 0; i<j; i++)
      ssyrk_t(A[j][i], A[j][j]);

   spotrf_t(A[j][j]);

   for (int i = j+1; i<N; i++)
      strsm_t(A[j][j], A[i][j]);
}
```

Figure 2.1: Example code of a Cholesky Decomposition in the OmpSs programming model

Figure 2.1 shows a code snippet of a Cholesky Decomposition programmed in the OmpSs programming model. Pragma annotations are used to identify and declare tasks. The keywords **in**, **out** and **inout** are used to specify input and output dependencies, corresponding to the read-only, write-only and read-write task data, respectively. This information is analyzed by the runtime to produce a task dependency graph that guides the execution, maintaining program correctness without the need for explicit synchronization.

2.2 Prefetching

Prefetching is a well-known and widely used mechanism to reduce memory access latency by moving data from off-chip memory into the cache hierarchy before it is requested. Prefetching can be done for instructions and/or data. In this section we provide a broad overview of some of the more widely used techniques for data prefetching and those that are more relevant to the work done in Chapter 4.

2.2.1 Traditional Prefetching

Prefetching schemes can be software, hardware-based or a combination of both. Hardware-based prefetching relies on a dedicated hardware structure called the prefetch engine. The prefetch engine, implemented within the cache hierarchy,[1] analyzes at runtime the stream of memory instructions and attempts to find patterns in order to predict future memory references. Prefetch engines can implement different algorithms with various complexities and various area requirements. Hardware-based prefetching is widely used in modern multicore processors, where it is common to find multiple prefetch engines implementing different algorithms in different cache cache levels.

The simplest form of prefetching, *one block lookahead* [25], fetches the next consecutive block $b + 1$ after a reference to block b. *Stride-based* prefetching relies on finding constant strides within the stream of memory accesses, either by lookahead into the instruction stream [26] or by using a program counter-indexed reference prediction table (RPT) to keep track of recent accesses [27]. Stride-based prefetching is highly effective for applications with linear data access patterns and is still used in modern multicore processors [5].

History-based prefetchers leverage the observation that memory access patterns tend to repeat within a program. *Correlation* prefetching in particular, attempts to correlate past memory behavior with future memory references. Markov prefetching is a form of correlation prefetching [28] where a state transition diagram is built with the history of memory accesses. Each state or node in the graph has a probability associated with a state transition that represents the likelihood that a memory reference will follow the target node. Transitions with a probability above a certain threshold are selected for prefetching. Nesbit and Smith [29] proposed using a global history buffer (GHB) to store past information more accurately than previous RPT-based schemes. The GHB is a FIFO-like structure that stores the cache miss history while eliminating stale data that can lead to useless prefetches.

More advanced prefetching schemes are able to dynamically modify the behavior of the prefetch engine(s). Srinath et al. [30] propose a scheme that uses dynamic feedback obtained at runtime to tune the aggressiveness of the prefetch engine based on its effect on performance. Jimenez et al. [31] propose a similar adaptive prefetching mechanism leveraging the capabilities of the programmable prefetch engine in the IBM POWER7 processor. Their algorithm dynamically adjusts the configurable prefetch parameters based on IPC variations during different application phases. The scheme we propose in Chapter 4 dynamically selects the best cache level to prefetch into based on estimated cache space available.

[1]This is known as processor-side prefetching. There are also proposals for memory-side prefetching where the prefetch engine is located in the memory controller.

Contrary to hardware-based prefetching schemes, software-based prefetching does not require additional hardware support, but requires executing special *prefetch* instructions. Most modern instruction set architectures (ISAs) provide some form of non-blocking *fetch* instruction that loads data from main memory into the cache hierarchy. To avoid potentially harming performance due to memory related exceptions, *i.e.*, page faults or segmentation faults, prefetch instructions are typically not allowed to cause exceptions. If the address referenced is incorrect and an error is incurred, the instruction is dropped. Prefetch instructions are inserted into the application code, usually by the compiler on an optimization pass [32], although it can also be manually done by programmers. In the x86 and ARM-v8 ISAs, prefetch instructions are considered hints and are not guaranteed to be executed [33, 34].

One of the main challenges of software prefetching schemes is finding the optimal position within the code to insert the prefetch instructions. This issue, also important for hardware-based schemes, is known as prefetch *timeliness*. Issuing the prefetch too early may evict useful data from the cache hierarchy (a problem known as *cache thrashing*), while doing it too late may not fully hide the latency of accessing off-chip memory. Gornish et al. [35] proposed an algorithm to find at compile time the earliest point in the code where prefetch instructions can be inserted, focusing specifically on array references within loops.

Mowry and Gupta [36] analyzed the impact of hand-inserted prefetch instructions and found that the performance improvement on applications with regular access patterns was significant. Prefetching on applications with extensive use of pointers and linked lists, on the other hand, was more complex and less successful. They further developed a compiler algorithm to automatically insert prefetch instructions in scientific codes [37]. Their algorithm analyzes the locality of memory references to find spatial and temporal reuse, and uses the number of iterations in a loop as a reference to find the scheduling point for the prefetch instructions. The timeliness of the prefetching scheme we propose in Chapter 4 relies on the runtime system's knowledge about the path of execution. By knowing *when* and *where* data is required, the runtime system can initiate prefetching with enough time to ensure, to a certain degree, that it will arrive before it is requested by the cores.

Hybrid prefetch schemes mixing hardware and software techniques have been proposed to benefit from the advantages of each method: the accuracy of the non-speculative software prefetching schemes and the performance potential of hardware-based prefetchers. Chen and Baer [38] explored a scheme where the compiler inserts prefetches for user-defined data objects of any size, fetching them into the second level cache. The hardware prefetch engine then works at cache line granularity and brings data into the first cache level, closer to the cores. They proposed defining a special control instruction that would enable or disable the hardware prefetcher with the goal of prefetching only during loops. This approach is similar to the scheme proposed in Chapter 4, but we leverage the runtime system to generate prefetch instructions using the dynamic information available during runtime instead of relying on the limited static information available to the compiler.

Wang et al. [39] proposed a hybrid scheme where the compiler encodes hints in load memory operations based on the presence of spatial locality or irregular data structures. The hints are propagated at runtime to the prefetch engine on the second cache level, that uses them to regulate its aggressiveness and reduce the bandwidth usage.

In addition to timeliness, two other metrics are used to evaluate the efficiency of a prefetching scheme: *accuracy* and *coverage*. Accuracy represents the percentage of prefetched cache lines that were actually referenced by the processor, and it is usually formulated as:

$$Prefetch\ Accuracy = \frac{Useful\ Prefetches}{Total\ Prefetches\ Issued}$$

Coverage represents the percentage of misses avoided due to prefetching, and it can be formulated as[2]:

$$Prefetch\ Coverage = \frac{Misses\ Eliminated\ due\ to\ Prefetching}{Total\ Cache\ Misses}$$

In general terms, these three metrics constitute three design points that must be balanced when designing a prefetch scheme. A very aggressive prefetcher may have very high coverage at the expense of many mispredictions and thus low accuracy. Alternatively, a conservative prefetch scheme may only issue prefetch requests when there is a high confidence that the block will be useful, thus having high accuracy but low coverage. Finding a good middle ground between them is a complex issue and can largely depend on the workloads.

The prefetching scheme we propose in Chapter 4 uses the information provided by the user about the task's input data to prefetch, and therefore, no speculation is required. In this manner, our scheme achieves 100% accuracy, as all the data prefetched is guaranteed[3] to be needed. Coverage, on the other hand, will depend on the percentage of data the task accesses that is declared by the user. In order to maintain program correctness all global data must be declared as either input or output, but the tasks are allowed to allocate extra local data, which is not declared in a pragma clause and will therefore not be prefetched.

Another important consideration for prefetch schemes is the *location* where data is prefetched into. Software prefetching schemes can use hints to decide which level of the cache hierarchy to place the prefetched data into. For example, the x86 ISA provides four different prefetch instructions: `PREFETCHT0, PREFETCHT1, PREFETCHT2` and `PREFETCHNTA`, that

[2]A different formulation can also be found in the literature as: Useful Prefetches / Total Cache Misses.
[3]Perfect accuracy depends on the user successfully identifying and specifying the task's input data.

indicate which levels of the hierarchy to prefetch into, while the ARM-v8 ISA `PRFM` instruction uses a *target* parameter to specify the prefetch destination. As discussed, the scheme we propose in Chapter 4 can dynamically decide the best cache level based on the task's input size and cache space available.

Hardware prefetchers place the fetched data in the cache level where the prefetch engine observing the memory access stream and issuing prefetch requests is located. It is also possible to place the prefetched data into a small *prefetch buffer* located next to the cache [40]. The advantage of doing so is that it avoids cache pollution and thrashing.

Cache pollution is caused by mispredicted blocks taking space in the cache; cache thrashing is caused by prefetched blocks – even when they are useful – evicting data that is still in use by the processor. The disadvantage of using a prefetch buffer, in addition to the extra die area required, is that it can either increase the cache access latency or waste power. If the prefetch buffer lookup is done only after the cache tag array lookup returns a miss, the operations are serialized and the access latency is increased. If the lookups are done in parallel, every cache access will consume power by doing a prefetch buffer look-up that may be unnecessary if the access hits in the cache.

2.2.2 Block Prefetching

Traditional software and hardware prefetching schemes work at the granularity of cache lines. In software-based schemes, this entails executing one prefetch instruction per cache line fetched, using valuable micro-architectural resources that could instead be used to execute instructions that make forward progress and thus introducing a non-negligible execution overhead [38]. In addition, prefetch instructions are interleaved in the code, increasing the size of the resulting binary and reducing the effective size of the instruction cache.

The benefit of prefetching large blocks of data instead of individual cache lines was first noted by Gornish et al. [35]. In their approach, the compiler performs static program dependence analysis on array references in nested loops, inserting a block prefetch command before the data is referenced. Wall [41] presented a study on the effect of different code optimizations on the memory subsystem, including software block prefetching using the `MOV` instruction. This approach consisted on manually inserting `MOV` instructions in the code, which, as author found out, may in some cases not work well with other compiler optimizations. Chen and Baer [38], as mentioned earlier, proposed a hybrid scheme where compiler-inserted prefetch instructions fetch blocks of memory corresponding to user-defined objects into the second cache level. The hardware prefetch engine then further brings data closer to the cores at cache line granularity.

ARM includes a block prefetcher in their Cortex-A8 and Cortex-A9 processors [42]. The Preload Engine (PE), as it is named, allows the user to load selected regions of memory into the L2 cache. The PE expects the programmer to add load directives by hand, requiring a good understanding of the code and some knowledge of the underlying architecture. The PE is attached to the cores, and is only able to direct the data transfers to the last level L2 cache. While this approach relies on compiler analysis or the programmer to manually insert prefetch instructions in the code, in our scheme the runtime system inserts them with minimal user intervention and based on dynamic information available at runtime.

Papaefstathiou et al. [43] propose a software prefetching and cache management mechanism for task-based programming models. They introduce a programmable prefetch engine that receives prefetch commands generated by the runtime system based on the data known to be used by the application. While their idea is similar to the scheme we propose, there are a few important differences.

First, whereas their proposal is an alternative to traditional hardware prefetchers, we propose a hybrid hardware-software prefetching scheme, where software prefetching brings data on-chip to hide the large DRAM latencies, and hardware prefetching moves the data closer to the cores.

Second, whereas Papaefstathiou et al. evaluate their approach using a simple in-order processor, our evaluation uses an advanced out-of-order processor that can hide on itself some memory latency. We therefore establish that the approach is also applicable to high-performance processors implementing aggressive instruction-level parallelism techniques where there is lower benefit from additional prefetching.

Third, they propose a prefetch engine per core, while our proposed prefetch engine may be shared by multiple cores, reducing chip area and power consumption. Additionally, grouping prefetch commands in a common engine allows for the coordination of priorities among the cores, and also allows us to introduce effective throttling mechanisms. Finally, while their approach prefetches only to the last-level cache, our scheme dynamically adapts to the state of the cache hierarchy and selects the cache level to prefetch into which is most beneficial at that time.

2.3 Heterogeneous GPU-CPU Architectures

We define heterogeneous architectures as systems composed of multicore processor(s) and one or many GPUs. The GPU has traditionally been a *discrete* board connected to the host machine through a system expansion bus, *e.g.* PCIe. Discrete GPUs contain their own pool of high-bandwidth memory, as well as their own cache hierarchy. Until recently, this memory

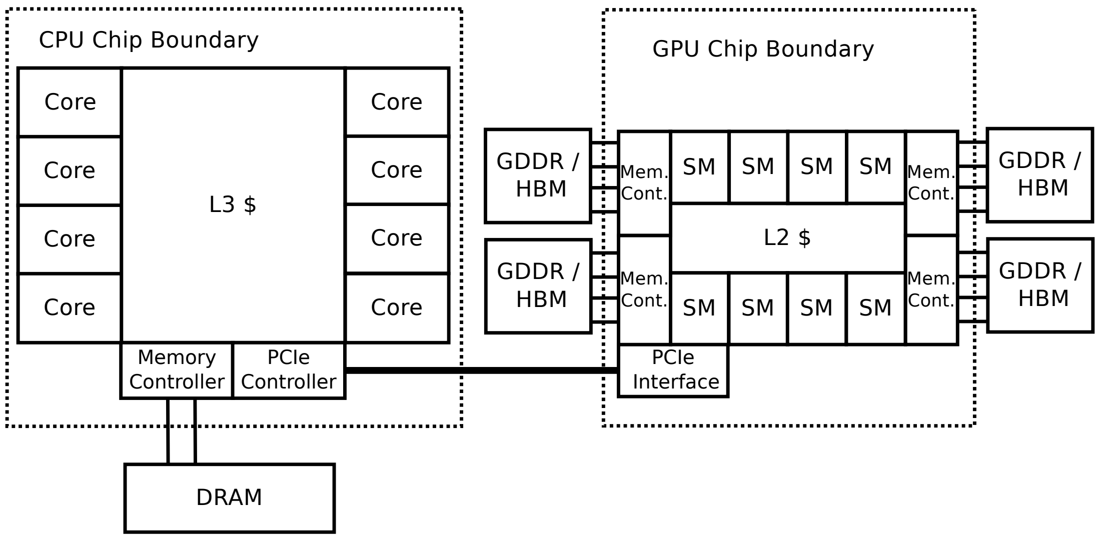


Figure 2.2: High level overview of a heterogeneous system composed of a multicore processor and a discrete GPU connected to the host through PCI-Express.

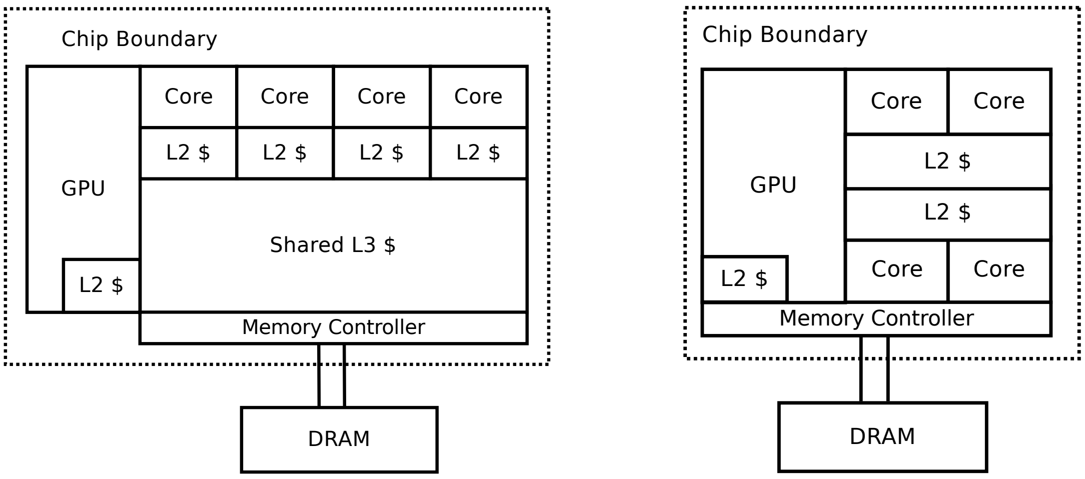
has been completely decoupled from the host's memory, residing in its own virtual address space and therefore not directly addressable by the host and vice versa.

In the traditional heterogeneous computational model, data allocated on the host must be explicitly copied to the device's[4] memory before it can be used. Explicit data transfers are done via direct memory access (DMA) operations using the DMA engine(s) found in the GPU. Figure 2.2 shows a high level overview of a heterogeneous system with a discrete GPU connected through PCIe. SM stands for Streaming Multiprocessor, NVIDIA's terminology for GPU cores; AMD's equivalent is compute units (CUs).

The current trend in heterogeneous system design is towards tighter coupling of GPU and CPU. From mobile and embedded chips [44, 45, 46] to desktop and laptop-oriented processors [47, 14], it is increasingly common to find architectures integrating the GPU on the same die as the CPU cores. In this design, the GPU is another element of the SoC, connected to the rest of the system through the network-on-chip (NoC). *Integrated* architectures tightly couple GPU and CPU cores, providing a shared pool of system memory, a unified virtual address space and even some degree of cache coherence [48, 16, 19].

On-die integration of GPU and CPU cores provides multiple benefits: a shared memory pool avoids explicit data movement and duplication; communication through the NoC instead of a dedicated interconnect (PCIe) saves energy and decreases latency; lower communication latency enables efficient fine-grained data sharing and synchronization. Consequently, an increasingly large body of research has been published on the benefits of heterogeneous computing on integrated systems [49, 50, 51, 52, 53, 54, 55].

[4]Host and device refer to the CPU and GPU respectively in NVIDIA's terminology.



(a) (b)

Figure 2.3: High level overview of two integrated heterogeneous architectures with different cache hierarchy designs: a) with a last-level cache shared between GPU and CPU. b) with no shared cache.

2.3.1 Resource Sharing on Integrated Architectures

Integrated systems require some degree of resource sharing between GPU and CPU, although implementations from different vendors differ in which ones. For example, both AMD and Intel processors use shared memory controllers to access off-chip memory [14, 47]. Intel also uses a unified ring bus as the NoC connecting GPU and CPU with the system agent and the memory controllers, while AMD implements two different bus paths for GPU and CPU to access the memory controllers.

The last-level cache (LLC) also differs in chips from Intel and AMD. While Intel processors integrate a shared LLC between GPU and CPU cores and off-chip memory, AMD's set of Accelerated Processing Units (APUs, AMD's terminology for integrated heterogeneous systems), on the other hand, completely separate the cache hierarchies of GPU and CPU. Similarly, NVIDIA does not implement a shared LLC in their line of integrated heterogeneous architectures [46].

Figure 2.3 shows a block diagram of two heterogeneous systems composed of a multicore processor and an integrated GPU. Figure 2.3a shows an architectural design similar to an Intel Haswell processor [47], where both GPU and CPU cores share a common level 3 cache. Figure 2.3b shows a high level overview of a processor similar to AMD's Kaveri [14], where there is no shared cache between GPU and CPU.

Some recent works have tackled the issues of resource sharing within heterogeneous architectures. Lee and Kim analyze the impact of LLC sharing between GPU and CPU [56].

They find that the multithreaded nature of GPUs allows them to hide large off-chip latencies by switching to different threads on memory stalls. In addition, they note that GPU workloads tend to stream through large amounts of data, showing a memory access pattern with little data reuse. Therefore, they conclude that caching is barely useful for such workloads, and argue that cache management policies in heterogeneous systems should take this into consideration. They propose TAP, a cache management policy that detects when caching is beneficial to the GPU application, and favors CPU usage of the LLC when it is not.

Mekkat et al. build on the same premise [57]. They use set dueling [58] to measure CPU and GPU sensitivity to caching during time intervals. With this information, they dynamically set a thread-level parallelism (TLP) threshold for each interval. The threshold determines after what amount of TLP the GPU's memory requests start bypassing the LLC. Their goal is to prevent the GPU from taking over most of the LLC space and depriving the cache-sensitive CPU of it.

Other works have explored the challenges of resource sharing within GPU-CPU systems. Ausavarungnirun et al. focus their study on the memory controller [59]. They find the high memory traffic generated by the GPU can interfere with requests from the CPU, violating fairness and reducing performance. They propose a new application-aware memory scheduling scheme that can efficiently serve both the bursty, bandwidth-intensive GPU workloads and the time-sensitive CPU requests.

Kayiran et al. consider the effects of sharing the NoC memory controllers [60]. They monitor memory system congestion and if necessary limit the amount of concurrency the GPU is allowed. By reducing the amount of active warps[5] in the GPU, they are able to improve CPU performance in the presence of GPU-CPU interference.

All these works analyze resource sharing within integrated GPU-CPU systems, but they perform their evaluation on multiprogrammed workloads where GPU and CPU execute different unrelated benchmarks. This methodology can shed light on some of the problems associated with resource sharing in heterogeneous architectures, but it is not able to provide any insight about the effect such sharing has in heterogeneous computations where GPU and CPU cores collaborate and share data. The goal of the work presented in Chapter 5 is to analyze how these heterogeneous algorithms are affected by sharing the LLC.

[5]Warp is NVIDIA's terminology for a group of threads running in lock-step on a core. AMD refers to the same concept as wavefront.

2.3.2 Heterogeneous Computing

The shift from graphics processing to general-purpose computing requires a set of new programming models and frameworks for GPU programming. These have evolved over time as GPUs have, introducing new features that improve programmability and simplify general-purpose computing on GPUs. The two programming models most used for heterogeneous computing are NVIDIA's CUDA and the open standard OpenCL.

The OpenCL programming model [61] offers support for heterogeneous computing between CPU cores and multiple accelerator-like devices, such as GPUs, field-programmable gate arrays (FPGAs) or digital signal processors (DSPs). It is an open standard contributed to by many different vendors, such as AMD, Apple, ARM, IBM and Samsung. Since the 2.0 specification, OpenCL includes features especially designed for integrated systems, such as Shared Virtual Memory (SVM) or system-wide atomic operations [62].

On a system supporting SVM features, the same pointer can be used indistinctly by the CPU and GPU, and coherence is maintained by the hardware as in a traditional SMP. System-wide atomic operations can be used to guarantee race-free code when sharing data through SVM. These atomic operations allow for fine-grained synchronization among computing elements, opening the door for heterogeneous applications that work on shared data structures and coordinate much faster than using previous methods.

CUDA is NVIDIA's proprietary programming model and API for general-purpose computing. Currently in its 8.0 version, CUDA has evolved to include many quality-of-life features that simplify the task of heterogeneous programming. Unified Virtual Addressing (UVA) was introduced in CUDA 4 [17], providing a common virtual address space between GPU and CPU that allows pointers allocated in one to be used directly by the other. CUDA 4 also introduced *zero-copy* memory, allowing the GPU to directly access pinned host memory through the PCIe interconnect.

In CUDA 6 NVIDIA introduced Unified Memory (UM) [63]. UM featured automatic data movement of memory regions allocated using `cudaMallocManaged().` In UM, managed memory pages are initially allocated in the GPU, populating the local page table. If data is initialized by the host, it is then migrated by the CUDA runtime transparently to the user, and on kernel launch, migrated back to the GPU for the computation [64].

This initial implementation of UM had several shortcomings: all managed memory modified by the host is copied to the GPU on kernel launch, even when it is not needed by the kernel; page frames are assigned immediately as memory is allocated and thus no memory oversubscription is possible; the managed region is limited to the size of the GPU's physical memory; data is migrated only at kernel boundaries and cannot be simultaneously accessed by GPU and CPU threads during the computation. Due to all these inefficiencies, the performance of UM is hardly able to compete with fine-tuned manual data movement [65, 66].

In 2016 NVIDIA unveiled CUDA 8 and the Pascal line of GPUs. CUDA 8 lifts these restrictions and allows host and device to concurrently access shared data, expands the managed memory region to cover both GPU and CPU physical memory and supports system-wide atomic operations [18]. The main feature that enables concurrent access to shared data in Pascal-based chips is demand paging.

In CUDA 8 memory is lazily-allocated, *i.e.*, the page frame is only reserved on first-touch access in either GPU or CPU memory. Since the GPU's page table does not contain the virtual to physical address mapping of pages allocated in the CPU, the first GPU access to one of such pages raises a page fault. GPUs are currently not able to context switch to execute a page fault handling routine like CPUs do, and therefore the GPU memory management unit handles the fault by forwarding it to the software runtime running on the host. The CUDA runtime can then migrate the page to the GPU or map it in the GPU's memory address space to be accessed directly through the interconnect.

UM and demand paging greatly simplify heterogeneous programming by relieving programmers from the burden of explicit memory management, relegating that job to the CUDA runtime and device driver. Unfortunately, the implementation currently found in Pascal-based GPUs, while convenient, is unable to match the performance of manual data movement via `cudaMemCpy()` operations. Processing GPU-initiated page faults incurs delays that not even the highly threaded design of GPUs can completely hide, causing underutilization of the compute units. The work presented in Chapter 6 tackles these inefficiencies and proposes an efficient mechanism to shared data between GPU and CPU.

2.3.3 Heterogeneous Memory Management

By heterogeneous memory management we refer both to: management of the memory subsystem on heterogeneous architectures, and management of hybrid memory designs combining memories of different technologies, *e.g.* traditional DRAM and 3D die-stacked or non-volatile memory.

2.3.3.1 Memory Management on Heterogeneous Architectures

On heterogeneous GPU-CPU architectures, one line of research has focused on the trade-offs between copying data to the GPU or accessing it directly through the interconnect [67, 68, 69]. The general idea is using heuristics to decide at runtime whether it is more beneficial to migrate data or to access it remotely based on metrics such as available bandwidth or total number of accesses. This topic is beyond the scope of this thesis; for our work in dynamic data movement we assume data is always migrated to the requester's local memory and never accessed remotely through the interconnect.

With the assumption of data migration on every remote access, Zheng et al. [70] propose a hardware/software approach to hide the latency of fault handling and automatic page migration as implemented currently in the Pascal family of GPUs by NVIDIA. They augment the GPU to support replaying fault-causing instructions, allowing the compute units to continue executing on a fault. In addition, they propose a page prefetching mechanism that speculatively requests and migrates pages to the GPU, aggregating multiple page migrations in one operation to amortize the costs of fault handling and DMA transfer.

Shahar, Bergman and Silberstein [71] take on a different approach, proposing a software layer that transparently enables address translation and paging on GPUs. Their goal is to provide a simple way to access files from the GPU, mapping files to the GPU's memory space and allowing easy access via regular pointers. Their work is specially interesting because they introduce a GPU-centric system to resolve page faults, moving away from the current implementation where page faults must be sent to the CUDA driver running on the host to be processed. This idea matches our intent of detaching the host from the process of GPU memory management whenever it is possible.

Kim et al. [72] propose a memory organization where the GPU's memory pool is used as a cache of CPU memory. They argue that using pinned host memory to remotely access data is inefficient as it causes multiple redundant memory transfers from host to device. Their scheme dynamically moves data at cache line granularity from host to device as it is referenced, keeping the working set of the kernel in GPU memory and taking advantage of its high bandwidth compared to remote accesses.

Similarly to the related work discussed in Section 2.3.1, the main difference between our work and all the related work presented here is that none of them consider collaborative heterogeneous applications with fine-grained data sharing between host and device. In all cases the issue is approached from the point of view of how best to manage GPU memory or maximize GPU performance, but always assuming data is consumed only by the GPU. In Chapter 6 we focus on collaborative computations where data migrates multiple times between host and device, as their data sharing pattern exerts more pressure in the demand paging scheme found in Pascal-based GPUs.

2.3.3.2 Management of Hybrid Memory Designs

Heterogeneous architectures with dedicated discrete GPUs already combine memories of different characteristics. The large majority of commodity processors use double data rate synchronous dynamic random-access memory (DDR SDRAM) for system memory, while GPUs integrate graphics DDR (GDDR) memory, a type of SDRAM specialized for higher bandwidth. Furthermore, the shift towards die-stacked memory technologies has seen a definite push on GPUs due to the significant larger bandwidth they provide. Therefore, the new family of GPUs coming from NVIDIA and AMD forgo GDDR and use some form of 3D-stacked high bandwidth memory (HBM) [73, 74].

Another form of hybrid memory design tightly couples memories of different technologies. For example, some proposals combine 3D die-stacked or non-volatile memory with traditional GDDR memory to obtain higher GPU performance and/or energy efficiency [75, 76]. On the CPU side, numerous works propose integrating a pool of 3D-stacked memory on-chip combined with DDR SDRAM-based system memory [77, 78, 79, 80, 81].

In particular, Chou, Jaleel and Qureshi proposed CAMEO [82] for a system where a high-bandwidth 3D die-stacked DRAM is integrated in a traditional symmetric multiprocessor with commodity off-chip memory. The stacked memory is placed between the last-level cache and off-chip DRAM and used as a high-capacity cache memory. Data is moved at cache line granularity between the system memory and the 3D-stacked DRAM cache transparently to the user and operating system, providing a high-bandwidth high-capacity last-level cache. The design of CAMEO serves as an inspiration for the memory organization we propose in Chapter 6.

2.3.4 Memory Consistency and Cache Coherence

Memory consistency models guarantee memory correctness on architectures using shared memory by providing rules about the behavior of load and store instructions. In broad terms, the semantics of a strict consistency model simplify programmability at the cost of performance. Relaxed consistency models allow compilers and hardware to perform memory reordering, increasing performance. This complicates the task of the programmer, since memory may need to be operated on with atomic operations or synchronized via fences.

The x86 ISA follows a relaxation of Sequential Consistency (SC) [83] called Total Store Order (TSO) [84]. In this model, loads following a store (in program order) can be executed before the store if they are to a different memory address. Although there is not much public information describing the memory consistency models followed by GPUs from the major vendors, they have been largely inferred to be relaxed models. One of such models is Release

Consistency (RC) [85]. RC enables many memory optimizations that maximize throughput, but is strict enough to allow programmers to reason about data race conditions. RC is the consistency model defined in the HSA standard [19], and it is followed in GPUs by vendors such as ARM [86] and AMD [87].

While the programmer must have knowledge about the consistency model followed by the architecture targeted in order to guarantee his or her parallel code does not show race conditions, and will therefore, execute correctly, coherence protocols are transparent to the user. Coherence protocols guarantee that all sharers of a datum always obtain the latest value written, and in most systems, are pivotal to maintaining memory consistency. Regardless of the protocol itself (*i.e.* MESI, MOESI, *etc.*), x86-based SMPs follow the coherence model Read For Ownership (RFO). In an RFO machine, cores must obtain a block in an exclusive state before writing to it. This scheme is effective for workloads that exhibit temporal locality and data reuse, where the cost of exclusively requesting blocks and the associated invalidation messages is amortized over time.

GPUs have traditionally exhibited a different memory access behavior, streaming through data with little data reuse. In addition, the high memory traffic generated by the large number of threads running concurrently exerts a high pressure in the memory subsystem, and any additional coherence traffic would only aggravate the problem. Because of this, GPUs implement very simple coherence mechanisms with private write-through write-combining L1 caches that can contain stale data [48, 88].

Recent work shows that the choice of consistency model minimally impacts the performance of GPUs [87]. While stricter consistency models and system coherence does not come for free, researchers are already working on solutions to solve the challenges faced [89].

We believe integrated systems will change the way we understand heterogeneous programming and change the characteristics of heterogeneous workloads. Stricter consistency models across a heterogeneous system will improve programmability and allow programmers to maintain the memory semantics they are used to on traditional SMPs. Therefore, the work on integrated heterogeneous architectures done in Chapter 5 is evaluated on a system implementing a TSO consistency model with RFO coherence across all computing elements.

Chapter 3
Methodology

This chapter presents the experimental methodology followed throughout this dissertation. We introduce the two architectural simulators employed as well as the benchmarks and metrics used in the evaluation of our proposals.

3.1 Simulation Infrastructure

The work on prefetching done in Chapter 4 is evaluated using TaskSim [90], an trace-driven, cycle-accurate simulator developed at the Barcelona Supercomputing Center that models an x86 multicore processor. Figure 3.1 shows the simulation workflow for TaskSim and OmpSs applications. We use the dynamic binary instrumentation tool PIN [91] to obtain memory traces; these traces are then combined with a trace of runtime system events. The combined trace is replayed by the simulator, interfacing during the simulation with the runtime system of the OmpSs programming model through a *bridge*. In this manner, the runtime system modified with our proposed prefetching scheme is natively executed during the simulation, and the dynamic behavior of the application run that depends on the architectural state (*e.g.* task schedule) is captured.

In Chapter 5 and Chapter 6 we use the gem5-gpu [92] simulator to study heterogeneous architectures. gem5-gpu is a cycle-level simulator that merges gem5 [93] and GPGPU-Sim [94]. Figure 3.2 shows the simulation workflow for heterogeneous applications on gem5-gpu. The GPU pipelines are simulated in detail by GPGPU-Sim, interfacing with an implementation of the CUDA Runtime provided by the gem5-gpu developers. The bridge between GPGPU-Sim and gem5 is a memory interface that transforms memory instructions issued by the GPU cores into memory instructions understood by the gem5 simulator. The memory interface injects transformed instructions into the gem5 memory subsystem modeling both GPU and CPU cache hierarchies, off-chip memories and interconnect fabric. Once the instructions are satisfied by the memory subsystem, the interface transforms and returns

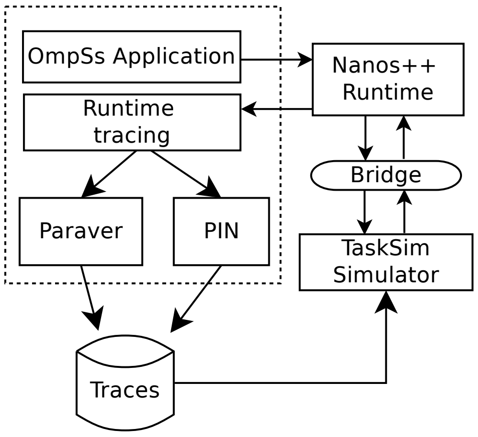


Figure 3.1: Simulation workflow for OmpSs applications and TaskSim.

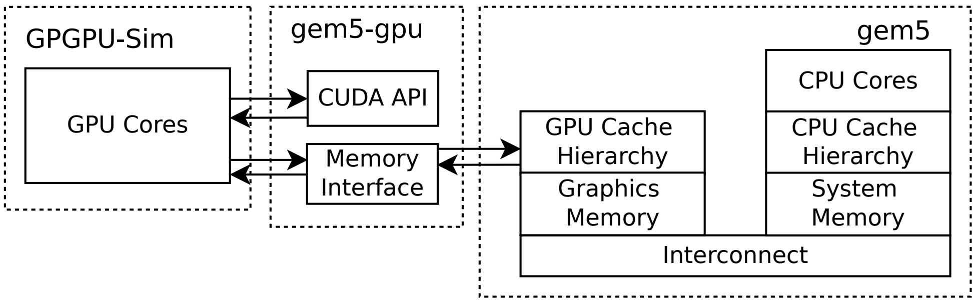


Figure 3.2: Simulation architecture of the gem5-gpu simulator.

the replies back to the GPU cores. We use gem5-gpu's full-system mode running the Linux operating system with kernel 2.6.28.

Chapter 5 presents an evaluation of different cache hierarchy organizations on a system where the GPU is integrated on-die with the CPU, while Chapter 6 focuses on architectures with a dedicated, discrete GPU. gem5-gpu can be configured to simulate both systems in its *fused* and *split* mode respectively.

In *fused* mode, the GPU is connected to the root crossbar as another element of the SoC. Both GPU and CPU share a unified virtual address space and both can directly access off-chip system memory. For virtual to physical address translation, the GPU uses the CPU's page table that is maintained by the operating system. GPU page faults are therefore resolved as CPU page faults, raising an interrupt and trapping into the OS to execute a fault handling routine. In the fused mode, the simulator provides full cache coherence between GPU and CPU. We use the MESI coherence protocol throughout the system, following the TSO consistency model.

In *split* mode, the GPU is simulated as a separate device board connected to the rest of the system through a PCIe interconnect. Initially, the GPU is in a different virtual address space and has its own page table and pool of memory. For our evaluation of a dynamic data movement scheme in Chapter 6 we modify the system to provide a unified virtual address space, similar to current GPUs from NVIDIA. In the baseline system, page faults initiated by the GPU are sent to the host to be handled. In our scheme, only the first GPU access to a page causes a fault, as explained in Section 6.3. In *split* mode there is no cache coherence between GPU and CPU. The CPU cache hierarchy uses the MOESI coherence protocol with a TSO consistency model, while the GPU uses a more relaxed consistency model and a simple valid/invalid coherence protocol. GPU's L1 caches are write-through and non-inclusive.

Chapter 4 and Chapter 5 present a power evaluation in the form of energy-to-solution. The results in both chapters were obtained with CACTI [95] version 6.5 configured with the parameters shown in Table 4.1 and Table 5.1 respectively.

3.2 Workloads

3.2.1 Adaptive Runtime-Assisted Block Prefetching

We evaluate the proposed block prefetching scheme using a set of scientific benchmarks including PBPI, a parallel implementation of Bayesian phylogenetic inference method for DNA sequence data [96], an implementation of the MD5 hashing algorithm and a set of kernels representing algorithms commonly found in scientific applications. The full list can be found in Table 3.1. All applications were compiled for x86-64 with the GCC compiler version 4.6.3 using the -O3 optimization flag. The results were validated to confirm the transformations done by the compiler do not alter program correctness.

We target scientific codes such as those used in the field of HPC. HPC applications usually operate on linear data structures and can therefore benefit both from our runtime-directed software prefetching scheme and from hardware-based prefetching techniques. Our runtime-directed prefetching scheme also works on applications with more irregular data structures as long as the tasks' input and output data is specified as described in Section 2.1.

An important aspect to consider in HPC applications is the granularity at which the work is divided. In order to fully exploit the cache hierarchy and improve performance, the programmer must choose an appropriate block or task size to work with. This decision is usually taken considering the size of the cache memories and the number of processing elements. To improve load balancing, it is usually desirable to split computation into small tasks, allowing the scheduler to keep all the cores busy at all times. On the other hand, working at a too small granularity adds non-negligible overheads in the form of thread or task creation. There is plenty of literature on the topic of how to best choose this parameter and the impact it has

Benchmark	Input size	Task creation	Task duration
Histogram	256KB	18μs	546μs
Matmul	128KB	14μs	631μs
Reduction	256KB	17μs	145μs
LU	128KB	16μs	1000μs
PBPI	200KB	13μs	114μs
Jacobi	258KB	15μs	245μs
MD5	512KB	14μs	2021μs

Table 3.1: Benchmarks evaluated, average task input size, average task creation overhead and average execution time per task.

on the overall system performance [97, 98, 99, 100, 101]. We create tasks as small as possible to obtain good load balancing and exploit L1 cache locality, while keeping the overhead of task creation relatively small over the total execution time.

Table 3.1 shows the average size of the inputs for each task, the average overhead of task creation and the average execution time per task. These numbers were obtained on a 16-core, dual-socket AMD Opteron 6128 machine running at a frequency of 2.4 GHz.

3.2.2 Heterogeneous Architectures

In Chapter 5 we evaluate a CUDA version of the Rodinia GPU benchmark suite. Rodinia GPU [102] is a benchmark suite widely used to evaluate GPUs. Benchmarks from Rodinia GPU follow the traditional heterogeneous computational model, where the host allocates and initializes the data, copies it to the device in bulk data transfers via `cudaMemCpy()` operations and launches a computational kernel. When the computation is completed, the results are then copied back to the host.

Since Rodinia benchmarks were designed for architectures with discrete GPUs, we modify them to make use of the characteristics of integrated systems. We thus remove all explicit data movement operations and substitute the allocations of data using `cudaMalloc()` calls with regular `malloc()` operations, leveraging the shared address space. Table 3.2 lists the Rodinia benchmarks evaluated and the input sets used.

As stated, Rodinia benchmarks follow the traditional model where the computation is largely done in the GPU and where data is shared between GPU and CPU in a coarse-grained manner only at kernel boundaries. One of the main goals of this thesis is to understand the implications on the memory subsystem when executing collaborative computations. Collaborative computations split algorithms into different steps that can be assigned to the compute unit best suited to execute them. Regions with high data or thread parallelism are sent to the

Table 3.2: Rodinia Benchmarks.

Benchmark	Short Name	Dataset
Backprop	RBP	256K nodes
Breadth-First Search	RBF	256K nodes
Gaussian	RGA	512 × 512 matrix
Hotspot	RHP	512 × 512 data points
LavaMD	RLA	10 boxes per dimension
LUD	RLU	2K × 2K matrix
NN	RNN	1024K data points
NW	RNW	8K × 8K data points
Particlefilter	RPF	10K particles
Pathfinder	RPA	100K × 10K data points
Srad	RSR	512 × 512 data points

GPU to take advantage of their massively parallel characteristics, while regions with low parallelism can be executed by the larger deeply-pipelined out-of-order CPU cores. These applications share data at fine granularities during the computation, using system-wide atomic memory operations to synchronize.

We use as well a set of collaborative benchmarks in the evaluation of Chapters 5 and 6. For the work in Chapter 5 we prepared a collection of collaborative benchmarks. They present different heterogeneous computation patterns and are summarized in Table 3.3.

Four benchmarks (DSP, DSC, IH, and PTTWAC) deploy concurrent CPU-GPU collaboration patterns. In these benchmarks, the input workload is dynamically distributed among CPU threads and GPU thread blocks[1]. DSP and DSC utilize an adjacent synchronization scheme, which allows CPU threads and/or GPU blocks working on adjacent input data chunks to synchronize. Each CPU thread or GPU block has an associated flag that is read and written atomically with system-wide atomic operations. Both DSP and DSC are essentially memory-bound algorithms, as they perform data shifting in memory. DSC deploys reduction and prefix-sum operations in order to calculate the output position of the elements.

IH carries out an intensive use of atomic operations on a set of common memory locations (*i.e.*, a histogram). Chunks of image pixels are statically assigned in a cyclic manner to CPU threads and GPU blocks. These update the histogram bins atomically using system-wide atomic additions. PTTWAC performs a partial transposition of a matrix. It works in-place; thus, each matrix element has to be saved (to avoid overwriting it) and then shifted to the output location. As each of these elements is assigned to a CPU thread or a GPU block, these need to coordinate through a set of atomically updated flags.

[1]Thread block is NVIDIA terminology for a group of threads that execute on the same core and can communicate via shared memory. AMD refers to them as work-groups.

Table 3.3: Heterogeneous Benchmarks Evaluated in Chapter 5.

Benchmark	Short Name	Field	Computation Pattern	Dataset
Breadth-First Search [103]	BFS	Graphs	Coarse-grain switching	NY/NE graphs [104]
DS Padding [105]	DSP	Data manipulation	Concurrent collaboration	2K $\times$ 2K$\times$ 256 float
DS Stream Compaction [105]	DSC	Data manipulation	Concurrent collaboration	1M float
FineGrainSVMCAS link [106]	LCAS	Synthetic benchmark	Fine-grain linked list	4K elements
FineGrainSVMCAS unlink [106]	UCAS	Synthetic benchmark	Fine-grain linked list	4K elements
Image Histogram [107]	IH	Image processing	Concurrent collaboration	Random and natural images (1.5M pixels, 256 bins)
PTTWAC Transposition [103]	PTTWAC	Data manipulation	Concurrent collaboration	197 $\times$ 35588 doubles (tile size = 128)
Random Sample Consensus [108]	RANSAC	Image processing	Fine-grain switching	5922 input vectors
Task Queue Histogram [103]	TQ	Work queue	Producer-consumer	128 frames

Table 3.4: Chai Benchmarks Evaluated in Chapter 6.

Benchmark	Short Name	Field	Computation Pattern	Dataset
Breadth-First Search	BFS	Graphs	Coarse-grain switching	NY/NE graphs
Bezier Surface	BS	Computer graphics	Concurrent collaboration	500 $\times$ 500 double (tile size = 16)
Canny Edge Dectection	CEDD	Image processing	Concurrent collaboration (data partitioning)	50 frames
Canny Edge Dectection	CEDT	Image processing	Coarse-grain switching (task partitioning)	50 frames
Image Histogram	HSTI	Image processing	Concurrent collaboration	Random and natural images (1.5M pixels, 256 bins)
DS Padding	PAD	Data manipulation	Concurrent collaboration	2K $\times$ 2K float (block size = 256)
Random Sample Consensus	RSCD	Image processing	Fine-grain switching (data partitioning)	5922 input vectors
Random Sample Consensus	RSCT	Image processing	Fine-grain switching (task partitioning)	5922 input vectors
Task Queue - Histogram	TQH	Work queue	Producer-consumer	128 frames
PTTWAC Transposition	TRNS	Data manipulation	Concurrent collaboration	197 $\times$ 35588 doubles (tile size = 64)

In BFS the computation switches between CPU threads and GPU blocks in a coarse-grain manner. Depending on the amount of work of each iteration of the algorithm, CPU threads or GPU blocks are chosen. CPU and GPU threads share global queues in shared virtual memory. At the end of each iteration, they are globally synchronized using system-wide atomics. LCAS and UCAS are two kernels from the same AMD SDK sample. First, a CPU thread creates an array which represents a linked list to hold IDs of all GPU threads. Then, in the first kernel (LCAS) each GPU thread inserts in lock-free manner their respective IDs into the linked list using atomic compare-and-swap (CAS). In the second kernel (UCAS) the GPU threads unlink or delete them one-by-one atomically using CAS.

RANSAC implements a fine-grain switching scheme of this iterative method. One CPU thread computes a mathematical model for each iteration, which is later evaluated by one GPU block. As iterations are independent, several threads and blocks can work concurrently. TQ is a dynamic task queue system, where the work to be processed by the GPU is dynamically identified by the CPU. The algorithm performs a histogram calculation of frames from a video sequence. Several queues are allocated in shared virtual memory. CPU threads and GPU blocks access them by atomically updating three variables per queue that represent the number of enqueued tasks, the number of consumed tasks, and the current number of tasks in the queue.

The benchmarks evaluated in Chapter 5 were the seed of the now publicly available Chai suite of collaborative heterogeneous benchmarks [109]. In Chapter 6 we use Chai to evaluate our proposed data migration scheme. Chai drops the *LCAS* and *UCAS* benchmarks and instead adds *Bezier* and *CEDD/CEDT*. In addition, we drop *DSC* because its behavior is very similar to *DSP* and provides the same insights.

From the new benchmarks, Bézier tensor-product surfaces are geometric constructions widely used in engineering and computer graphics [110]. Chai's implementation divides the surface into four-sided tiles, each of which is computed by a GPU block or a CPU thread. The size of the GPU blocks is the same as each tile, so each output point is computed by one GPU thread. CPU and GPU threads access a shared list of tiles to obtain the next tile to process, thus work is dynamically assigned at runtime and system-wide atomic operations are used to coordinate.

CEDD implements a Canny Edge Detection algorithm widely used in image processing. In it, multiple frames of a video are processed through four stages, implemented as four different computational kernels. Chai provides two implementations of the algorithm. CEDD partitions the input set and assigns frames either to the GPU or to the CPU. In this implementation, each frame is entirely processed by one or the other. CEDT partitions the algorithm by task, where the two first processing steps are done by the CPU and the remaining two by the GPU. Similarly, Chai provides two implementations of RANSAC with two different partition schemes. RSCD splits the input dataset assigning iterations to either GPU or CPU threads. RSCT partitions the algorithm by tasks, where the sequential fitting stage is done by CPU threads and the evaluation of the model is done by GPU blocks.

The full list of benchmarks used in Chapter 6 is shown in Table 3.4 with the input sets used. For both the Rodinia benchmarks and the collaborative benchmarks evaluated in Chapters 5 and 6, we select and evaluate only the region of interest, skipping initialization (memory allocation, input file reading, *etc.*) and clean-up phases.

3.3 Metrics

We use several metrics to evaluate the performance of our block prefetching scheme. The most straightforward metric is execution time. Since our proposed scheme can (and we argue that it should) be used in conjunction with the hardware prefetch engines of modern processors, first we find the best hardware prefetch configuration for every benchmark. We use that configuration as the baseline, and show execution time for all benchmarks normalized to it.

The goal of a prefetch scheme is to bring useful data into the cache hierarchy, thus improving cache hit rate. We therefore show cache hit rates for all three levels of the cache hierarchy. Another metric commonly used to measure the performance of the memory subsystem is average memory access time (AMAT). We calculate AMAT as:

$$AMAT = AccessTime_{L1} + MissRate_{L1} * MissPenalty_{L1}$$

were

$$MissPenalty_{L1} = AccessTime_{L2} + MissRate_{L2} * MissPenalty_{L2}$$

and

$$MissPenalty_{L2} = AccessTime_{L3} + MissRate_{L3} * MissPenalty_{L3}$$

and $MissPenalty_{L3}$ equals the average time to access off-chip main memory.

In addition, we use energy-to-solution to evaluate whether our scheme has a positive or negative impact on total energy usage. The results are normalized to the configuration with the best hardware prefetcher only.

Chapter 5 presents a comparison between two cache hierarchy designs. We aim at showing the impact of having a shared last-level cache among GPU and CPU cores. Again, the most straightforward metric is execution time. Unless stated otherwise, we show execution times normalized to the configuration with private LLC. To compare the effect of sharing the LLC on cache hit rates, we show the hit rates for both shared and private LLC configurations. For the private configuration we calculate LLC hit rate as:

$$Hit\ rate = \frac{Hits\ LLC_{CPU} + Hits\ LLC_{GPU}}{Access\ LLC_{CPU} + Access\ LLC_{GPU}} * 100$$

To understand why sharing a LLC may have a positive effect on performance, we look at the timing to perform system-wide atomic memory operations. In gem5-gpu atomic operations are performed with a read-modify-write (RMW) instruction. This operation is divided in two steps, a first load of the cache line with exclusive state and a following write with the new value. The time to perform the initial load is therefore representative of the time required to obtain and *lock* the cache line, and hence of the time to perform the atomic operation.

We use the time to perform the load to evaluate the impact that sharing the LLC has on performing system-wide atomic operations. In addition, we use instructions-per-cycle (IPC) of both GPU and CPU to understand how it impacts the performance of GPU and CPU separately. Finally, in Chapter 5 we analyze the energy implications of a shared LLC by measuring energy-to-solution with a breakdown of its different contributors within the memory hierarchy, *i.e.*, DRAM and the three cache levels.

To motivate the work done in Chapter 6 on dynamic data movement, we analyze the current scheme of demand paging found in NVIDIA GPUs. We show a breakdown of all page faults raised during the execution of a set of benchmarks, differentiating those raised by the CPU for pages located in GPU memory, those raised by the GPU on a first access to a page located in CPU memory and for those which the GPU has already migrated at some point to its memory but are currently located in CPU memory. In addition, to show the inefficiency of migrating full memory pages, we show the percentage of data that is migrated back and forth without being referenced.

In the evaluation presented in Chapter 6 we show execution times normalized to a configuration resembling the demand paging scheme found in NVIDIA GPUs. To understand the impact of reducing the granularity of migrations, we show the total number of migrations for various migration sizes, normalizing the result to number of migrations with the smallest possible size of one cache line. We also provide an analysis on the impact of varying the interconnect round-trip time. We present execution time for different latencies and migration sizes. For each latency-migration configuration we normalize the result to the execution time with the baseline demand paging configuration and that same link latency.

Chapter 4
Adaptive Runtime-Assisted Block Prefetching

4.1 Motivation

The processor-memory performance gap still remains a significant source of performance loss in modern multicore processors. Throughout the years, many mechanisms have been developed that can alleviate the problem by hiding some or all the latency of accessing off-chip memory, including: non-blocking caches, out-of-order execution and data prefetching. Data prefetching in particular is a widely used technique that triggers the movement of data from off-chip memory into the cache hierarchy before it is needed.

Software-based prefetch schemes rely on executing special prefetch instructions, usually inserted in the code by the compiler on an optimization pass. Most implementations of prefetch instructions found in modern ISAs fetch one cache line per instruction. This can lead to a non-negligible execution overhead and has a negative impact on the instruction cache [38]. Prefetching blocks of data of variable size with a single instruction is a good solution to this problem. Several works in the literature have proposed block prefetching schemes, some relying on compiler analysis [35], others on manual insertion of prefetch directives in the code [41] and others using a runtime system to guide the prefetch engine [43].

While all approaches can be successful in some circumstances, compiler analysis is still limited and manually inserting prefetch instructions in the code is a difficult and time-consuming endeavor. Using a runtime system to guide prefetching, on the other hand, is a simple and efficient way of performing block prefetching. A runtime system can see further into the future than current compilers are able to, has dynamic information of the application and requires minimal user intervention.

In particular, the runtime system of task-based programming models is specially well suited to guide prefetching, as it has all the required information to make effective block prefetching, knowing accurately *when*, *where* and *what*.

- **When:** the runtime system knows *when* a task is going to execute because it builds a task dependency graph, and its scheduler guides the execution flow.
- **Where:** the runtime system knows *where* data will be needed because it knows in advance which core will execute each task.
- **What:** the runtime system knows *what* input data is required by each task, as indicated by the programmer via pragma directives.

All this information puts the runtime system in a advantageous position to perform data prefetching while alleviating the main drawbacks of traditional prefetching schemes. Knowing *when* and *where* data is needed allows the runtime to adjust the timeliness of the prefetch requests and to prefetch directly into the cache of the core that needs the data, while knowing *what* data is needed avoids speculation and thus the risk cache pollution due mispredictions. In addition, if the runtime system is provided a map of the cache hierarchy, it can dynamically adjust the prefetch destination, placing data into a lower cache level if necessary to avoid cache thrashing.

In this chapter we propose a hybrid prefetching scheme that combines a runtime-assisted block prefetcher with existing hardware-based prefetch schemes. The runtime system guides a prefetch engine in bringing on-chip large blocks of data. Once the data is on-chip, traditional hardware prefetching mechanisms are used to bring data closer to the CPU at cache line granularity. The runtime system leverages its information about application schedule to decide when to start prefetching. In addition, it compares the task input data and cache sizes to dynamically select the best prefetch destination for each task without displacing the working set of the currently executing task.

4.2 Target Architecture

Our scheme targets a multicore processor following a SMP design. Figure 4.1 shows a high-level overview of the architecture with the addition of the multicore data transfer engine (MDTE). The MDTE is a small DMA-like controller that receives the prefetch commands generated by the runtime system and initiates the fetch operations from main memory. Section 4.4 provides the implementation details of the MDTE and explains how it interfaces with the cache hierarchy.

We evaluate the proposed prefetching scheme on a multicore processor with three different configurations of 4, 8 and 16 cores. In each case, each core has private L1 and L2 caches.

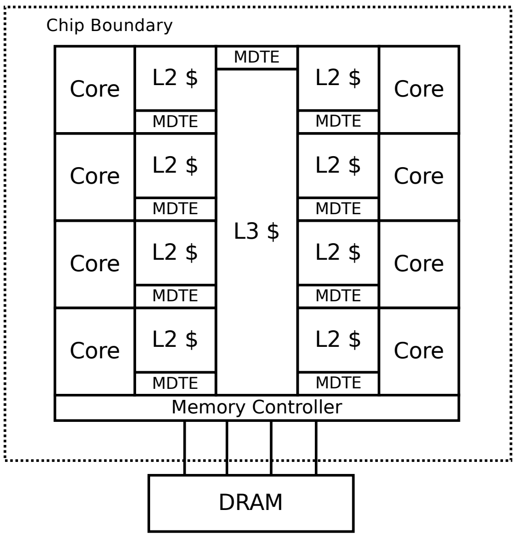


Figure 4.1: High-level overview of the target multicore architecture with private and shared MDTEs.

All the cores are connected through a crossbar to a shared L3, which is connected to off-chip main memory. The MDTE can be placed next to a core's L2 or the shared LLC. If placed next to a private cache it will only process prefetch commands from that core. If placed next to the LLC it can receive and process prefetch commands from every core. While our scheme would work on a system integrating only the shared MDTE, ideally we also want private MDTEs to let the runtime system decide which one to use in each case.

4.3 Block Prefetching

In order to avoid the overhead of executing one prefetch instruction per cache line and leveraging the information about tasks' input data available to the runtime system, we implement a special *prefetch command* instruction. Prefetch commands are similar to normal prefetch instructions but reference a contiguous block of memory. They accept two parameters to indicate a starting address and data size. They are generated by the runtime system based on the input data of a task and have unrestricted length.

In order to enable the runtime system to issue prefetch commands we extend the ISA with the following user mode instruction:

$$prefetch\langle L\rangle\ \langle rb\rangle, \langle rs\rangle$$

where rb is the register holding the base address of the block to be prefetched, rs is the register holding the size of the block in bytes, and L takes the value of the cache level to which the prefetch command is to be sent. In this manner, the instruction $prefetch2\ \langle r1\rangle, \langle r2\rangle$

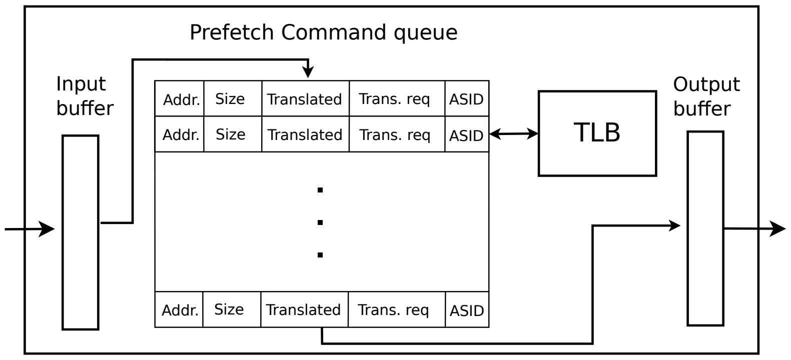


Figure 4.2: Multicore Data Transfer Engine components.

would send a prefetch command with the address indicated in r1 and the size indicated in r2 to the data transfer engine corresponding to the core's L2 cache. In order to send a prefetch command to the shared L3 cache level, the runtime system would issue the instruction $prefetch3\ \langle r1\rangle, \langle r2\rangle$.

If the runtime system has not been provided with a map of the cache hierarchy and there is no L3 cache in the system, the instruction is ignored. In our implementation, one bit in the instruction word is enough to specify whether the prefetch instruction targets the L2 or the L3 cache. We do not support block prefetching into the L1 cache because our experiments show that it is not large enough to prefetch with such granularity (more details on Section 4.5.1).

Prefetch commands initially reference virtual addresses, but since the physical pages they map to may not be contiguous in memory, they need to be split at page boundaries. Splitting prefetch commands and address translation is performed in the MDTE (see Section 4.4 for details).

4.4 Multicore Data Transfer Engine

The MDTE is a programmable DMA-like controller that receives and processes the prefetch commands generated by the runtime system. Figure 4.2 shows its design. The main components are:

- An input buffer to store received prefetch commands until they are queued.
- A prefetch command queue where commands are inserted in FIFO order. Each command in the queue can prefetch up to one memory page. Each entry in the queue holds the starting address, size, address space identifier (ASID), a translated bit and a translation requested bit.

- A Translation Lookaside Buffer (TLB) to speed up address translation.
- An output buffer to store translated commands until they are sent to memory.

The MDTE reads the input buffer for new commands. When a new command is received, it is split into page-contained commands and enqueued in the prefetch command queue. New commands are discarded when the queue is full. The commands received contain virtual addresses that need to be translated. There are two main advantages to delaying the translation until the command arrives at the MDTE: first, if address translation were to be done at the core's MMU, a prefetch command for a big block of data (*e.g.* a few megabytes) would be split into a large number of page-sized prefetch commands. These would have to travel to the corresponding MDTE, increasing traffic on the interconnect and wasting bandwidth. Second, address translation at the MMU's is in the critical path. Translations for prefetch commands would delay the translation of demand requests, further degrading performance.

The MDTE contains a TLB to speed up address translation and reduce the traffic caused by the translation requests. The impact of adding these TLBs is not significant since they need not be very large (see Table 4.1). We use a TLB directory to minimize the overhead of TLB shootdowns [111]. Once a translation response is received, the prefetch command is updated and moved to the output buffer. Interrupts and exceptions can modify the virtual to physical address mapping, rendering the prefetches useless. In these situations we flush the TLB and the entries in the prefetch command queue which translation has been requested, as well as the translated commands from the output buffer.

On every cycle at most one request will be issued, either a prefetch command or a translation request. Commands from the output buffer are sent to their target cache where they are issued one cache line at a time in round robin fashion. These prefetches coexist with hardware-based prefetch requests but are much less time sensitive, hence the need for some form of arbitration. See Section 4.5.2 for more details.

4.5 Runtime-Assisted Prefetching

Prefetch commands are generated by the runtime system for the tasks' input data as specified by the user via pragma clauses. With the tasks' input and output the runtime system builds a task dependency graph that represents the flow of data. Figure 4.3 shows the task dependency graph created by the runtime system for the Cholesky Decomposition shown in Section 2.1.

This graph is used by the runtime scheduler to guide the path of execution guaranteeing that data dependencies among tasks are respected. It also enables the runtime system to start prefetching with enough time to guarantee, to a certain degree, that data is present in the

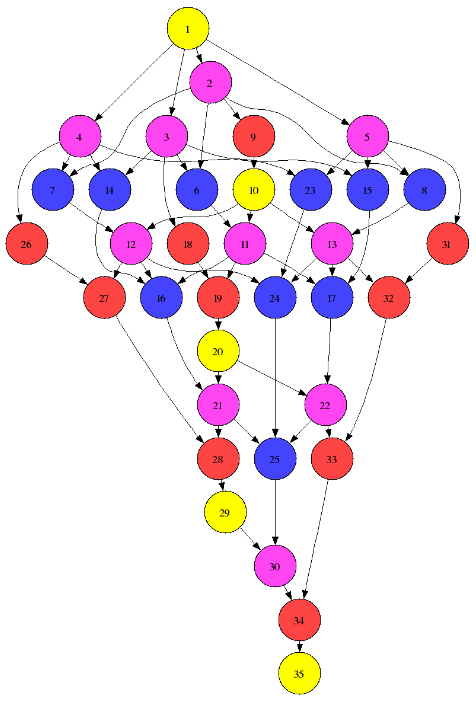


Figure 4.3: Task graph generated by the runtime system for a Cholesky Decomposition. The numbers indicate the task creation order and the colors the task type.

cache hierarchy by the time it is needed. Prefetch timeliness depends on the size of the input data and the time required to execute the task. Our evaluation shows that the average time to execute a task is significantly larger than the time required to prefetch a task's input data for sensible task sizes, see Table 3.1. Thus, prefetching a task's input data is triggered right before the execution of the preceding task begins.

Figure 4.4 shows the sequence diagram for an example of data prefetching directed by the runtime system of the OmpSs programming model. When the currently executing task A completes, the runtime scheduler uses the task dependency graph to obtain the next two tasks that can be executed: B and C. The runtime system generates a prefetch command for the input data of task C. The prefetch command, an operation in the order of tens of assembly instructions that entails a negligible overhead compared to the cost of running the runtime scheduler, is executed by the core before task B starts executing.

Task B begins executing while data for task C is being prefetched, overlapping data movement and computation. In addition, task C is pinned to the hardware thread executing task B, disabling work stealing and guaranteeing that task C is scheduled to execute on the core which caches hold the prefetched data. By doing so the runtime system implicitly applies an affinity-based scheduling policy, allowing for simpler scheduler algorithms.

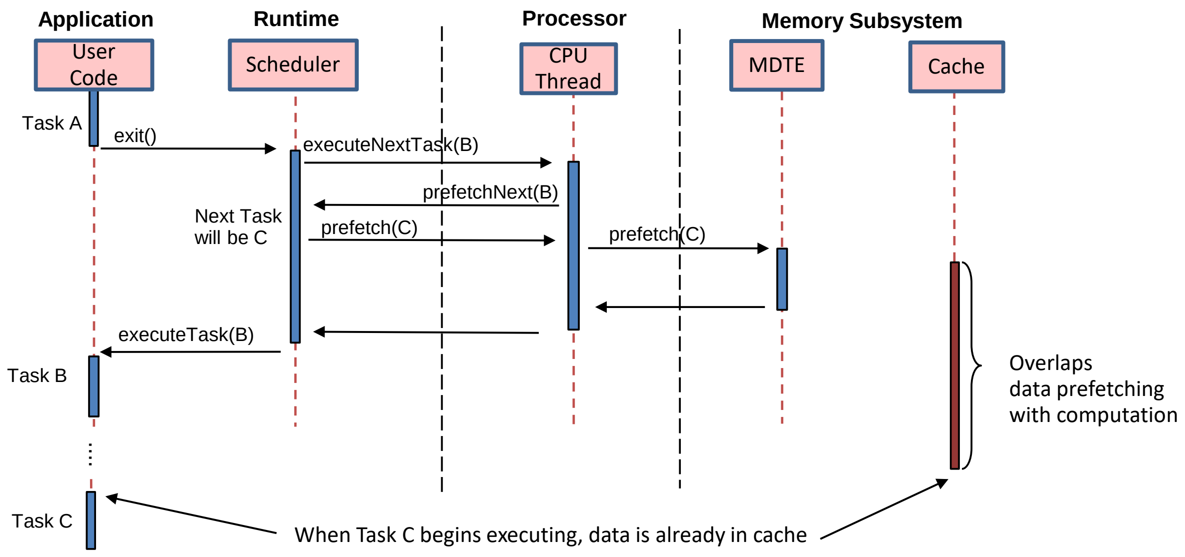


Figure 4.4: Sequence diagram of runtime-assisted prefetching on OmpSs.

4.5.1 Adaptive Destination

As shown in Figure 4.1, we propose integrating the MDTE logic in two locations, a private per-core MDTE and a shared MDTE that can be used by all cores. The private MDTEs will always forward the translated commands to the private cache they are attached to, and the shared MDTE to the LLC. Thus, another important aspect to determine is *where* to send the prefetch commands to, *i.e.*, the prefetch destination.

It is always desirable to place the prefetched data as close to the cores as possible without hurting the performance of the current task. Although the runtime system does not know exactly the content of each cache, it has knowledge of the input data used by each task. Using that information and a map of the cache hierarchy it is able to approximate where the prefetched data can be placed without evicting the working set of the current task. In this manner, the runtime system can dynamically decide the best prefetch destination before issuing the prefetch command.

Our experimental evaluation shows that L1 caches are typically too small for block prefetching, as they cannot hold the prefetched data without evicting the working set of the current task. Hence, the runtime initially attempts to prefetch data into the private L2 cache. Once the runtime system estimates the L2 cache cannot hold more data without evicting the current task's working set, it directs the remaining prefetch commands to the shared MDTE.

Figure 4.5 summarizes the algorithm used by the runtime system to decide the prefetch destination. The amount of data that can be placed in the L2 is calculated as:

$$Capacity_{L2} = Size_{L2} - Input_{curr} - PrefData_{next}$$

```
$PrefData_{next}$ = 0
while $Input_{next}$ > 0:
        $Capacity_{L2}$ = $Size_{L2}$ - $Input_{curr}$ - $PrefData_{next}$
        if $Capacity_{L2}$ > 0 then:
                L2 prefetch up to $Capacity_{L2}$ bytes
                increase $PrefData_{next}$
                decrease $Input_{next}$
        else
                L3 prefetch $Input_{next}$ bytes
        endif
```

Figure 4.5: Algorithm used by the runtime system to decide the prefetch destination.

where $Size_{\mathrm{L2}}$ is the size of the L2 cache, $Input_{\mathrm{curr}}$ the size of the input data for the task currently executing and $Input_{\mathrm{next}}$ for the task that will be executed next. $PrefData_{\mathrm{next}}$ represents the amount of data already prefetched from the next task.

As an example, Figure 4.6 shows the prefetch destination for two executions of the same benchmark with two different cache configurations. In this example, for simplicity, all tasks have 160 KB of input data.

The caches are assumed to initially hold stale data, so the input set for task 1 is always placed in the L2. On a system with a 128 KB L2 cache, only 128 KB of data fit; the remaining 32 KB are then prefetched into the L3 cache. When the runtime system begins prefetching for task 2, the L2 is full with tasks' 1 working set, therefore the 160 KB of data are prefetched into the L3. This behavior repeats until the end of execution. On a system with a 256 KB L2 cache, the 160 KB of input data from task 1 are initially placed on the L2. When the runtime system begins prefetching for task 2, 96 KB of its input data are prefetched into the L2 and the remaining 64 KB into the L3. On this configuration the working set of the currently executing task co-exists with a portion of the following tasks input data.

The L3 cache is assumed to be large enough to hold the working set of each of the executing tasks plus the prefetched data. As discussed in Section 3.2, it is usually desirable to divide the computation into small tasks to improve load balancing. Table 3.1 shows the average task input data size for our workloads, and Table 4.1 the configuration parameters of the simulated architecture. This shows that even for tasks with the largest input data size, the L3 cache is large enough to fit all the required data. Since the runtime system can be informed of the characteristics of the memory hierarchy, if the ratio of task input data to last-level cache size were to change, it would be trivial to modify the runtime system to stop prefetching when necessary.

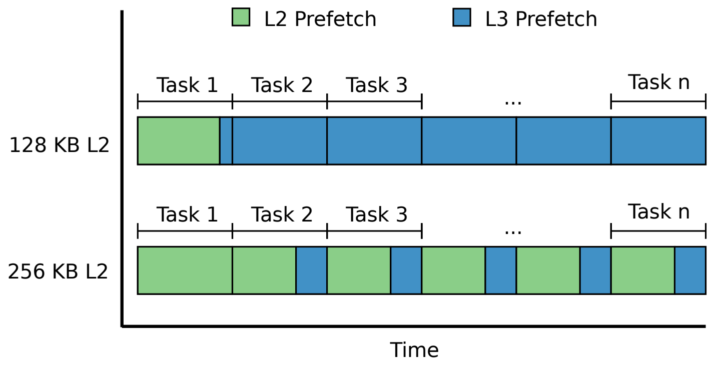


Figure 4.6: Prefetch destination of the input data for each task for two runs with different L2 configurations. Input data size: 160 KB.

4.5.2 Coordinating Hardware and Software Prefetch with Demand Loads

The main goal of our mechanism compared to previous prefetching work is to bring data on-chip at a coarser granularity (blocks vs cache line) with the help of the runtime system, and combine it with other traditional hardware and/or software prefetching mechanisms to move data closer to the core, *i.e.* the L1 or L2 caches. Unfortunately, prefetching has potentially a high cost in terms of bandwidth usage and network contention, specially if multiple and simultaneous prefetching mechanisms are used. Throttling policies [112] can be used to coordinate them, slowing or even stopping completely one of the prefetch engines in order to maintain fairness or avoid contention on shared resources.

Our implementation takes into account some priority considerations to ensure that requests in the critical path are always processed first. The first consideration is that demand requests generated by the CPU are always prioritized over prefetch requests. This ensures no prefetch instruction will delay a CPU request. Also, software prefetches are not as time sensitive as hardware prefetches, as the data prefetched is only required for the next task which is usually hundreds of thousands or millions of cycles in the future (see Table 3.1). Hardware prefetch engines analyze the stream of accesses and generate requests for data needed in the near future, and therefore are prioritized over the runtime-generated prefetches.

In addition, while demand requests are always prioritized, in-flight prefetches may still stall the memory subsystem if any of the hardware structures becomes full (input buffers, MSHR queues, *etc.*). We apply a simple throttling policy to deal with this issue. Any time that a cache level is unable to process a new request, we stop issuing new prefetch requests in that cache until demand requests can again be successfully processed. By doing so we give time to the in-flight requests to complete and we avoid getting the hardware structures filled with new prefetch requests that would further stall demand requests.

Table 4.1: Memory hierarchy configuration parameters.

Parameter	Value	Parameter	Value
Cache (L1/L2/L3)		DRAM DIMM	
Size (KB)	32/256/2048 per core	Data rate (MT/s)	1600
Latency (cycles)	2/12/45	Burst length	8
Associativity	2/8/16	CL/RCD/RP/RAS (cycles)	11/11/11/34
MSHR entries	8/32/8 per core		
MDTE (L2/L3)		Memory Controller	
TLB size	16/16	Access queue size	128
Prefetch queue size	256/1024	Number of DIMMs	4

4.6 Methodology

In order to evaluate the performance of our prefetching scheme we use the simulation infrastructure described in Section 3.1. We model the timing of an out-of-order processor, cache hierarchy, interconnection network and off-chip memory. The configuration parameters of the cache hierarchy are shown in Table 4.1. The cache line size is 128 bytes divided into 16 sub-blocks of 8-bytes each for all cache levels. All caches are inclusive, non-blocking and implement an LRU replacement policy. The bandwidth of all on-chip network links is 8 bytes per cycle with a latency of 3 cycles.

The MDTEs are implemented as described in Section 4.4 and configured using the parameters shown in Table 4.1. For energy estimations we use CACTI version 6.5 with the memory parameters specified in Table 4.1 and technology parameters based on ITRS predictions for a 32nm technology.

We evaluate our block prefetching scheme using the seven scientific benchmarks shown in Table 3.1. As stated earlier, we run simulations of a multicore processor with three different configurations of 4, 8 and 16 cores. Each core has private L1 and L2 caches, and all the cores share the L3 LLC. The LLC is multi-banked, with an 8MB bank per each 4 cores. As we increase the number of cores we add an additional memory controller per each additional LLC bank to sustain the extra traffic generated by the cores.

4.6.1 Hardware Prefetching

First we explore the effectiveness of the standalone hardware prefetchers for each of the benchmarks. We implemented and evaluated two commonly used hardware prefetching schemes: *Next-line* is the basic one block lookahead described in Section 2.2.1 that prefetches the next N lines after a cache miss. *Stride* is a reference prediction table-based stride prefetcher [27] that looks for regular strides among memory references from the same static instruction. We explored a range of values for the prefetch degree and found N=2 to be

Benchmark	Best HW pref.
Histogram	L1 Nextline + L2 Stride
Matmul	L1 Stride
Reduction	L1 Nextline
LU	L2 Nextline
PBPI	L1 Nextline + L2 Stride
Jacobi	L1 Nextline + L2 Stride
MD5	L1 Nextline + L2 Stride

Table 4.2: Best standalone hardware prefetch configuration.

optimal for both schemes and the simulated architecture. We evaluated all the benchmarks with all possible combinations of these prefetching schemes, *e.g.* only L1 stride, L1 stride and L2 nextline, L1 and L2 stride, *etc.* Table 4.2 shows which hardware prefetching scheme obtained the best performance for every benchmark.

We then repeat the experiments executing the benchmarks with all the hardware prefetch permutations possible, but combined with our runtime-assisted prefetching scheme. For all benchmarks but one, the hardware prefetch configuration that performs best standalone is also the best configuration in our hybrid hardware + software approach. The exception is LU, where every hardware + software configuration degrades performance by at least 5% over no prefetching. For the rest of this evaluation, we use the best standalone hardware prefetch configuration shown in Table 4.2 as the baseline for each benchmark. This configuration is labelled as *HW* on the figures. The configuration with the best hardware prefetcher and our proposed runtime-assisted prefetching scheme is labelled as *HW+MDTE*.

4.6.2 Compiler-Based Software Prefetching

We aim to compare our scheme to other traditional software prefetching techniques. We therefore compile every benchmark with the GCC flag `-fprefetch-loop-arrays`. With this optimization flag the compiler attempts to insert ISA-specific prefetch instructions into loops that traverse large data arrays.

As stated before, our hybrid approach combines runtime-assisted block prefetching with other traditional prefetching mechanisms that move data closer to the cores once it is brought on-chip by the MDTE. Thus, we not only use the compiler-based prefetch scheme to compare our proposal against, but we also evaluate the impact of combining both. We first execute the benchmarks compiled with the prefetch flag in conjunction with every hardware prefetcher and select the best performing. This configuration is labelled as *HW+SW* in the figures. We then take this configuration and combine it with our runtime-assisted block prefetcher (labelled as *HW+SW+MDTE*).

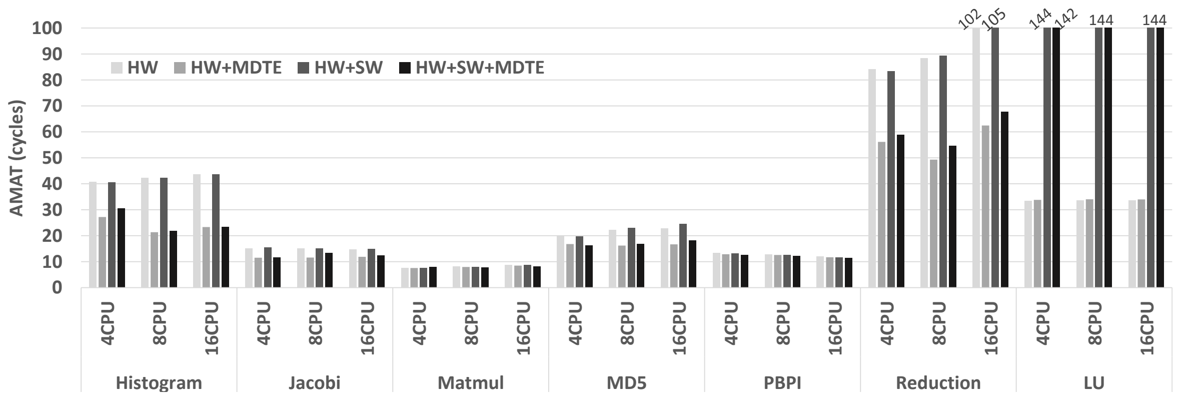


Figure 4.7: Average memory access time in cycles.

4.7 Experimental Evaluation

In this section we evaluate our proposed runtime-assisted prefetch scheme by looking at average memory access time (AMAT), cache hit rates and execution time. We also evaluate the power implications of using our prefetching scheme, including the additional power consumption caused by the MDTEs.

4.7.1 Average Memory Access Time

Figure 4.7 shows the AMAT for all the benchmarks on the various prefetch configurations discussed. For six of the seven benchmarks the MDTE is able to reduce AMAT. As expected, applications that display a high AMAT (even with hardware prefetching) benefit more from our software block prefetcher. In particular, *Jacobi*, *MD5*, *Reduction* and *Histogram* obtain on the 8 core configuration a reduction in AMAT of 18%, 28%, 48% and 49% respectively over the execution with the best hardware prefetcher only.

The benefit obtained by our hybrid scheme is limited to a 5% AMAT reduction for *PBPI*. The reason is that the AMAT for this application is already very low (20 cycles) with no prefetching mechanism, and it is even further reduced to 14 cycles by the hardware prefetcher. Since the latency of our L2 caches is 12 cycles and we model out-of-order cores that can hide some of that latency, the benefit attainable is very limited. A similar effect can be seen in *Matmul*, with an AMAT on the hardware prefetch configuration standalone of 8 cycles that our runtime-assisted prefetcher is not able to reduce. *LU* shows some interesting results, where using the compiler-based software prefetching scheme increases AMAT more than 4 times over the hardware prefetcher standalone configuration. In addition, we can see how our runtime-assisted prefetcher also increases AMAT slightly (2%). In order to better understand the reason we look at cache hit rates.

4.7.2 Cache Hit Rates

Figure 4.8 shows hit rates for the three cache levels. We can see why *Matmul* barely obtains any AMAT reduction with the runtime-assisted block prefetcher. Our implementation of matrix multiply uses the BLAS library and applies an optimization known as blocking, where the matrix is split in small blocks that can be computed concurrently. The block size adapts to the size of the L1 cache, and thus the benchmark has a 99.9% L1 hit rate. In addition, we can see how the L3 cache hit rate is also close to 99% on the hardware prefetch standalone configuration. With almost no misses in the cache hierarchy and corresponding off-chip accesses, our prefetching scheme cannot further improve performance. The memory access pattern of *Matmul* is very regular and therefore the stride-based hardware prefetcher is able to successfully predict and prefetch most future memory references.

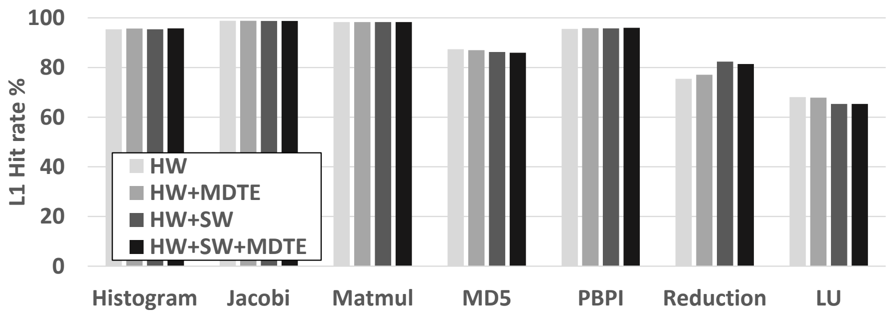
Figure 4.8a shows the cause of the large spikes in AMAT observed in *LU* when using the compiler-based software prefetch scheme. The compiler-inserted prefetch instructions are evicting useful data from the L1 cache due to bad timing, reducing the hit rate and consequently increasing the AMAT. We can also see why our scheme slightly increases AMAT. *LU* factorization uses blocking as well, with a block size of 128 KB that fits comfortably in the L2 cache the benchmark achieves near 100% L2 cache hit rate on the hardware prefetcher standalone configuration. Our runtime-assisted prefetcher reduces L2 hit rate slightly by evicting data from the current task's working set. The runtime system is not correctly identifying the available space in the cache and is therefore prefetching more data that actually fits. The cause is that the benchmark allocates private static data that is not declared as an input, affecting the heuristics that calculate the optimal prefetch destination.

Nevertheless, due to the inclusive cache hierarchy, the L2 data evicted by the prefetcher remains in the L3 cache, and in the end, our scheme successfully prefetches almost all data used by the benchmark, achieving a 99% L3 cache hit rate.

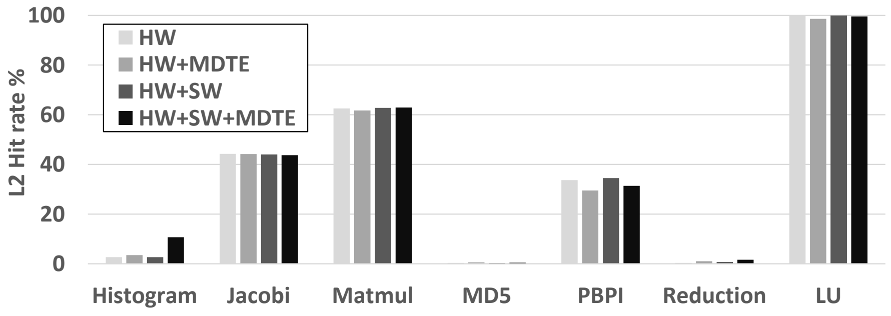
Overall, Figure 4.8 shows how our runtime-assisted prefetch scheme is able to bring on-chip most of the data used by the benchmarks. All benchmarks but one achieve 90% L3 cache hit rate or higher. The exception is *PBPI*, where our scheme achieves a 79% L3 hit rate only and barely improves the hardware prefetcher standalone configuration. The reason is that similarly to *LU*, the benchmark allocates a significant amount of local static data, which the runtime system does not know about and can therefore not prefetch.

4.7.3 Execution Time

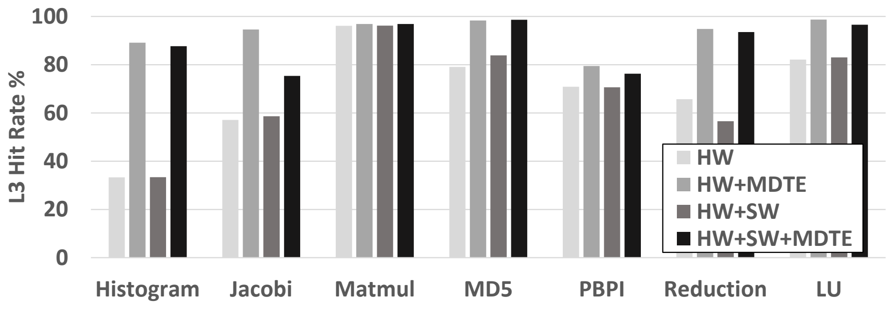
Finally, we evaluate execution time to see how the AMAT and cache hit rate changes affect performance. Figure 4.9 shows the speedup for all benchmarks with the various prefetch



(a) L1 hit rate.



(b) L2 hit rate.



(c) L3 hit rate.

Figure 4.8: Cache hit rates for all benchmarks on a 8 core system.

configurations over the execution with the best hardware prefetcher standalone. As stated in Section 4.6.2 we first evaluate the compiler-based software prefetch configuration in conjunction with the best hardware prefetcher for each benchmark. The results indicate that using the GCC prefetch flag produces mixed results depending on the benchmark.

The AMAT increase caused by the low L1 hit rate in *LU* translates into a performance drop of 46% on a 4 core system when using compiler-based prefetching. On the other hand, *Reduction* sees a significant 44% speedup thanks to an increased L1 cache hit rate. Additionally, compiler-based prefetching provides a small improvement in *PBPI* and a slight performance loss on *MD5*. The rest of the benchmarks see almost no variation compared to a hardware prefetch-only configuration.

As explained in GCC's documentation [32], compiling with the prefetch flag may gener-

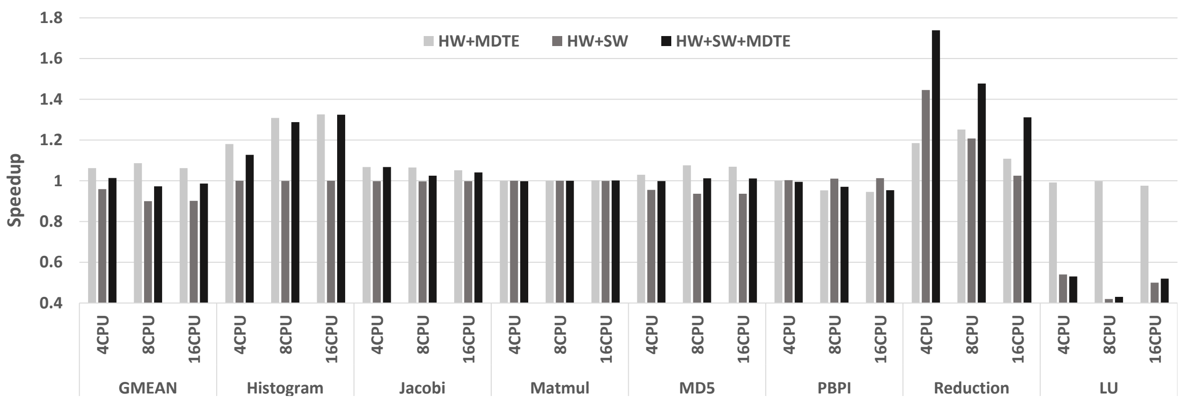


Figure 4.9: Speedup over the baseline configuration with hardware prefetching only.

ate better or worse code and is highly dependent on the structure of loops, and therefore it is an unreliable mechanism to consistently improve performance. Still, our proposed technique is designed to work in conjunction with any other fine-grained prefetching mechanism, so it is at the discretion of the user whether to use GCC-based software prefetching or not.

Combining hardware prefetching with our runtime-assisted software scheme produces more consistent results. In the 4 core system, our hybrid hardware + MDTE configuration obtains a 19% speedup over execution with the best hardware prefetcher standalone for *Histogram* and *Reduction*, and a more modest 7% on *Jacobi*. These gains are clearly associated to the substantial increase to L3 cache hit rate that our scheme provides.

PBPI and *Matmul* do not improve over the hardware prefetch configuration standalone. As discussed earlier, this is due to the low AMAT the benchmarks already show on the baseline configuration. Our scheme is not able to further reduce AMAT and hence provides no performance gains. *LU* sees a slight performance degradation due to the eviction of useful data from the L2 cache.

When combined with compiler-based software prefetching, our scheme is able to provide a significant 73% speedup for *Reduction* over the baseline hardware prefetch configuration. On this configuration our runtime-assisted prefetch scheme brings data on-chip, significantly increasing L3 cache hit rate. The compiler-inserted prefetch instructions further improve performance by prefetching data into the L1 that the hardware prefetcher alone cannot.

On average, the hybrid hardware + MDTE configuration obtains a 7% speedup over the baseline in the 4 core chip. Although the configuration including compiler-inserted prefetch instructions may perform best in some benchmarks, in others such as *LU* the performance drop is considerable, and overall the best results are obtained with hardware prefetching and our runtime-assisted prefetching scheme.

In the system with 8 cores we double the number of L3 banks and memory controllers to better support the bandwidth requirements of the extra cores. In this context our hybrid prefetching scheme shines obtaining a 30% and 25% speedup in *Histogram* and *Reduction* respectively, with an average of 9% for all benchmarks.

The configuration with compiler-inserted prefetch instructions experiences a large performance loss on *Reduction* over the system with 4 cores. The reason is that even with the additional memory controllers, the number of prefetch requests generated by the compiler saturates the interconnect network and memory controllers, diminishing the benefits obtained. *PBPI* suffers a small performance degradation because, as explained before, block prefetching does not provide any benefit over an already low AMAT, and because, as in the case of *LU*, the overhead caused by the prefetch requests traveling through the memory subsystem is non-negligible.

These results are maintained on the 16 core configuration with one exception: *reduction* loses about 10% performance on our hybrid hardware + MDTE configuration. The reason is that the LLC saturates with the increased number of requests and our throttling mechanism stops all prefetching. More complex throttling policies could be applied to reduce the impact of the increased traffic, and are left for future work.

4.7.4 Energy Consumption

Prefetching is usually considered a trade-off between performance and energy consumption, especially on speculative hardware-based prefetchers [113]. Yet, our proposed runtime-assisted prefetching scheme brings only data known to be needed, and the additional hardware required to support our block prefetcher has an almost negligible cost in area and power. In order to evaluate whether our scheme has a positive or negative impact on energy consumption, we analyze the static and dynamic power consumption of all benchmarks on all the different prefetch configurations.

Figure 4.10 shows energy-to-solution for each benchmark, with a breakdown of the different sources of energy consumption: dynamic power for each cache level and off-chip DRAM, as well as the total energy derived from static power in the system. We show results for all the prefetch configurations, each represented by a stacked bar. Results are normalized to the energy-to-solution on the hardware prefetch standalone configuration. The increase in power caused by the MDTEs has been included in the dynamic power of the cache level they are attached to, *i.e.*, L2 for the private MDTEs and L3 for the shared.

The results show how energy consumption is dictated primarily by static power, and therefore by execution time. Thus, the additional power consumption caused by the MDTEs is offset by the reduced execution times obtained using our hybrid prefetching scheme. This translates into energy-to-solution improvements of 10% on average for all benchmarks on the 4 core configuration. In all but two benchmarks we consume less energy by using our hybrid scheme compared to hardware prefetching only. On the 8 core configuration, *Reduction* and *Histogram* obtain a 13% and 15% decrease in energy-to-solution compared to the best hardware prefetch configuration standalone, with an average of 12% for all benchmarks. As expected, *PBPI* and *LU* see slight increases in energy consumption, *e.g.* 6% and 1% respectively on the 8 core configuration. The hybrid prefetching scheme is not able to further reduce execution times beyond what the hardware prefetcher is able to, and therefore there is no energy-to-solution reduction.

4.8 Summary and Concluding Remarks

In this chapter we propose a hybrid hardware + software block prefetching scheme. Prefetching is a technique widely used to reduce the processor–memory performance gap by bringing data from the high-latency off-chip memory into the cache hierarchy in advance.

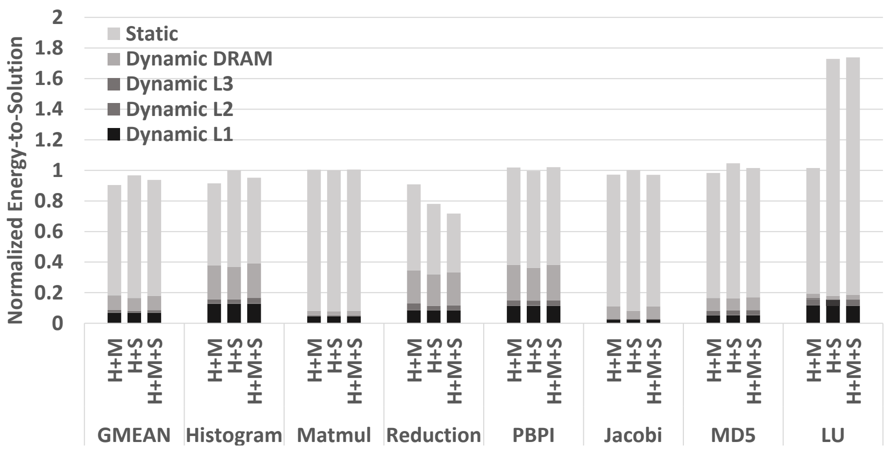
We have demonstrated that by using a runtime system to guide a block prefetch engine we effectively increase cache hit rates and hence reduce the average memory access time. This approach is simpler and more robust than manually inserting prefetch instructions in the code or relying on complex compiler analysis, a mechanism we have shown that provides mixed results, significantly degrading performance in some cases.

By using a runtime system with knowledge of the upcoming task schedule and memory referenced, we prefetch only data that the programmer states will be needed, avoiding cache pollution. In addition, we let the runtime system leverage this information to dynamically decide the best prefetch destination and avoid cache thrashing. Our proposal is especially efficient for memory-sensitive applications, but does not harm compute-bound applications.

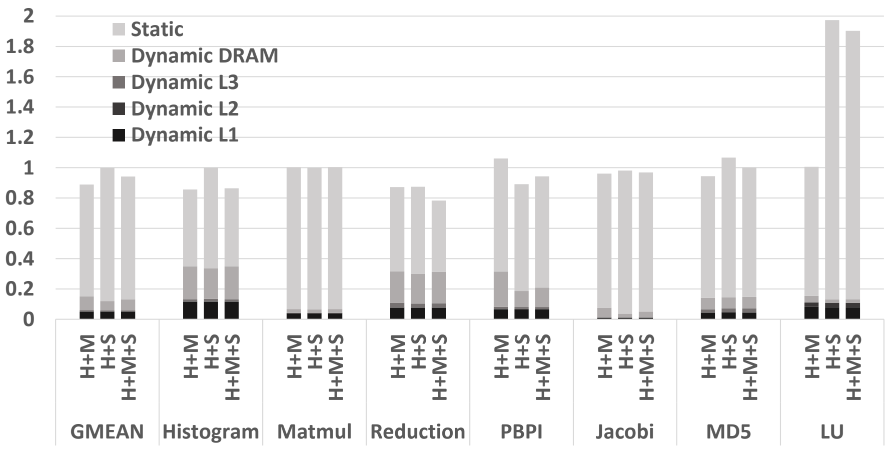
We show that the best results are obtained with a hybrid prefetch scheme combining our runtime-guided block prefetcher with other traditional hardware and software prefetching techniques that manage locality at cache line granularity. Our runtime-assisted block prefetcher brings large chunks of data from off-chip memory into the L2 or L3 caches, while the other prefetchers move the data closer to the cores, further reducing memory access times. For best results, we apply basic throttling to coordinate the prefetchers and reduce the overhead caused by the prefetch engines.

The evaluation on a set of scientific workloads shows that our hybrid prefetching scheme is able to obtain up to 32% performance improvement with an average of 9% compared to the baseline configuration with a hardware prefetching scheme only. The performance benefits offset the increased power from the extra hardware and the increase in dynamic power caused by prefetch activity, leading to a reduction of up to 18% with an average of 3% in energy-to-solution.

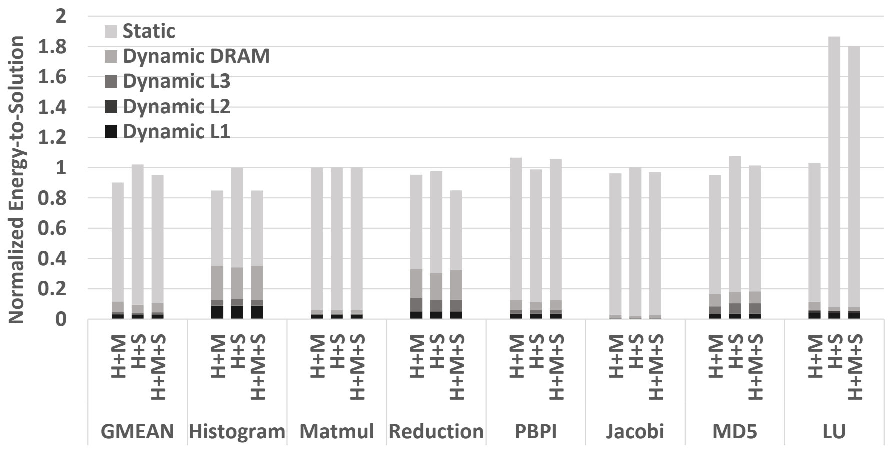
The experimental evaluation acknowledges our hypothesis that leveraging the information available to the runtime system of task-based programming models provides a perfect opportunity for efficient data prefetching. In addition, it shows that a hybrid prefetch scheme combining the best characteristics of software and hardware-based prefetchers is the most effective way of managing data prefetching in a multicore system.



(a) 4 cores.



(b) 8 cores.



(c) 16 cores.

Figure 4.10: Energy-to-solution normalized to the execution with the best hardware prefetcher standalone. From left to right for each benchmark: hardware + MDTE prefetch (H+M), hardware + software prefetch (H+S) and hardware + MDTE + software prefetch (H+M+S).

Chapter 5

Last-Level Cache Sharing on Integrated Heterogeneous Architectures

5.1 Motivation

Heterogeneous systems have become commonplace in the field of HPC. GPUs are widely used as accelerators for their enormous computing power and energy efficiency. While most GPUs used in HPC are still in a separate chip connected to a host machine through a computer expansion bus such as PCIe, the trend is towards tighter coupling of host and device.

In particular, on-die integration of GPUs and general-purpose CPUs has become the norm from desktop computers [14, 47] to mobile and embedded chips [46, 44]. This tighter coupling of GPU and CPU cores allows for seamless sharing of data structures and low-overhead synchronization, improving programmability and making heterogeneous computing more accessible. Yet, although on-die GPU integration seems to be the current trend among the main microprocessor manufacturers, there are still many open questions regarding the architectural design of these systems.

An important issue that has not yet been fully explored is resource sharing within these integrated heterogeneous architectures. While resource sharing within homogeneous SMPs is a well known and extensively studied problem, integrating computing elements with such widely different characteristics as GPU and CPU cores have presents new challenges. Thus, we are starting to see some work analyzing the effect of sharing on-chip resources such as the last-level cache [7, 57], the memory controller [114, 59, 60], or the network-on-chip [115]. Most of these works start with the premise that GPU and CPU applications exhibit different characteristics (spatial vs. temporal locality) and have different requirements (high bandwidth vs. low latency), and therefore careful management of the shared resources is necessary to guarantee fairness and maximize performance. In their evaluation, the authors use workloads composed of a mix of GPU and CPU applications running concurrently.

While using multiprogrammed workloads to evaluate resource sharing can give insights into some of the challenges of GPU-CPU integration, we believe it is not representative of future HPC workloads. The tight integration of CPU and GPU cores enables features such as a unified virtual address space and hardware-managed coherence, increasing programmer productivity by eliminating the need for explicit memory movement. GPU and CPU cores can seamlessly share data structures and perform low-overhead synchronization via atomic operations. In this manner, algorithms can be divided in smaller steps that can be executed on the device they are best suited for (*i.e.*, data parallel regions on the GPU or serial/low data parallelism regions on the CPU). These collaborative computations fully leverage the capabilities of integrated GPU-CPU systems, and their data sharing patterns will have implications on the shared resources that need to be understood.

The design of the cache hierarchy on integrated GPU-CPU systems varies from vendor to vendor and even among families of products from the same vendor. Even such a fundamental decision as whether to provide a shared cache level between GPU and CPU is not agreed upon by the major vendors. Intel chips since the Sandy Bridge family integrate the GPU on-die with the CPU cores [47], and include a shared L3 cache connected to the same ring bus as the GPU and CPU cores. AMD, on the other hand, completely separates the cache hierarchies of GPU and CPU in their APUs (integrated heterogeneous systems in AMD terminology) [14], as does NVIDIA in their Tegra line of integrated systems [46]. In this chapter we move a step forward towards understanding the effect of sharing the LLC on such architectures, and in particular when executing collaborative computations. Our goal is to provide guidelines for the design of the cache hierarchy of future integrated architectures, as well as for applications to best benefit from these.

5.2 Methodology

In order to analyze the effect of sharing the LLC we evaluate the set of heterogeneous GPU-CPU workloads detailed in Section 3.2 with the two cache hierarchy designs depicted in Figure 5.1. Configuration a) has separate, *split* L3 caches for GPU and CPU; memory requests from one can only go to the other through the directory. Configuration b) has a unified, *shared* L3 cache that both GPU and CPU can access directly and equally. In the evaluation performed in this chapter we analyze the the effect that sharing the LLC as in configuration b) has for heterogeneous computations where GPU and CPU collaborate and share data.

We simulate a four core CPU and an integrated GPU composed of four Fermi-like SMs grouped in two clusters of two. Considering NVIDIA Tegra X1 is composed of two SMs [46], this configuration is our best guess as to how the next generation of heterogeneous

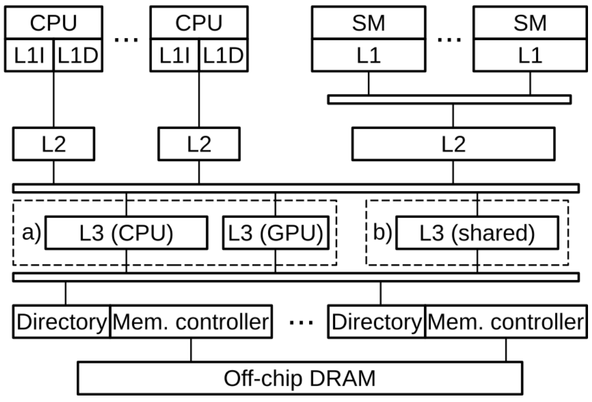


Figure 5.1: Integrated heterogeneous architecture with a) separate L3 caches for GPU and CPU, and b) a shared L3 LLC.

systems may be. The CPU's L1 and L2 caches are private per CPU core. Each GPU SM has a private L1, connected through a crossbar to the shared L2, which is itself attached to the global crossbar. In configuration a) the system has two L3 caches private to GPU and CPU; configuration b) shows a unified L3 that can be used by both.

Table 5.1 lists the configuration parameters of the system evaluated. The LLC size listed refers to the shared configuration. For the split configuration we partition the cache, giving 1/8 to the GPU and 7/8 to the CPU. We follow current products from Intel and AMD where the ratio of GPU-to-CPU cache size is between 1/8 and 1/16 [47, 48]. We evaluated different split ratios from 1/2 to 1/16 and saw similar trends among them.

Unfortunately, partitioning the LLC in this manner and directly comparing the results would not provide a fair evaluation. The additional cache space available to both CPU and GPU cores in the shared configuration may affect the results if the benchmarks are cache sensitive. To isolate the gains that are caused by faster communication and synchronization from those that are due to the extra cache space available in the shared configuration, we also run all the benchmarks with an extremely large, 32-way associative LLC of 1 GB total aggregate size. In this configuration, even on the split configuration the working set of most benchmarks fits in both private caches, and therefore the gains when the cache is shared cannot be attributable to the extra space available.

In both cases, the LLC(s) run in the same clock domain as the CPU. This allows us to present a fair comparison by setting the same access latency on both configurations, albeit providing a conservative estimation of the benefits of LLC sharing. All caches are write-back and inclusive with a LRU replacement policy. Cache line size is 128 bytes across the system. The NoC is modeled with gem5's detailed Garnet model [116]. Flit size is 16 bytes for all links; data message size is equal to the cache line size and fits within 9 flits (1 header

Table 5.1: Simulation Parameters for the Integrated Heterogeneous Architecture.

CPU	
Cores	4 @ 2 Ghz
L1D Cache	64kB - 4 way - 1ns lat.
L1I Cache	32kB - 4 way - 1ns lat.
L2 Cache	512kB - 8 way - 4ns lat.
GPU	
SMs	4 - 32 lanes per SM @ 1.4 Ghz
L1 Cache	16kB + 48kB shared mem. - 4 way - 22ns lat.
L2 Cache	512kB - 16 way - 4 slices - 63ns lat.
LLC and DRAM	
LLC	8MB - 4 banks - 32 way - 10ns lat.
DRAM	4 channels - 2 ranks - 16 banks @ 1200 MHz
RAS/RCD/CL/RP	32 / 14.16 / 14.16 / 14.16 ns

+ 8 payload flits). Control messages fit within 1 flit. The power results in Section 5.3.2 were obtained with CACTI version 6.5 [95] configured with the parameters shown in Table 5.1.

Our simulation infrastructure, detailed in Section 3.1, simulates a discrete heterogeneous architecture with global coherence and a unified virtual address space, allowing both GPU and CPU to access data allocated by the host using the same addresses.

5.3 Experimental Evaluation

This section presents the experimental evaluation of the effects of sharing the LLC as described in Section 5.2. We first evaluate the kernels from Rodinia GPU listed in Table 3.2. As discussed in Section 3.2, we modify the benchmarks to use regular pointers, leveraging the shared address space. Next, we evaluate a set of collaborative benchmarks that fully make use of the characteristics of integrated heterogeneous systems.

Collaborative heterogeneous benchmarks split the computation into steps that are assigned to either GPU or CPU cores, share data at fine granularities during the computation and synchronize via system-wide atomic operations. These benchmarks are a better representation of what we believe will be heterogeneous computations in the future. Their data sharing patterns tax the memory subsystem in different ways than traditional heterogeneous kernels do, and thus analyzing them provides insights that can be useful for the design of future heterogeneous architectures.

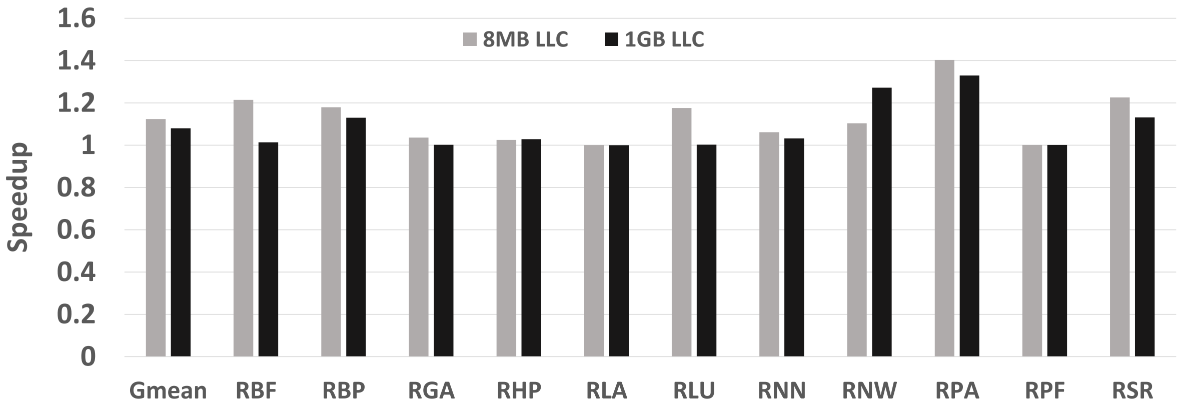


Figure 5.2: Speedup for Rodinia benchmarks with a shared LLC. For every shared LLC size the results are normalized to the private LLCs configuration with that same size.

5.3.1 Rodinia

As stated in Section 3.2.2, Rodinia benchmarks have minimal interaction between GPU and CPU. The one interaction all benchmarks share is in the allocation and initialization of data by the host prior to launching the computation kernel(s). Therefore, on a shared LLC configuration, if the working set of the benchmark fits within the LLC, the initial GPU memory requests after the kernel launches will hit in the LLC, avoiding an extra hop to the CPU's private LLC with the corresponding coherence traffic. The performance impact of finding a *warm* cache depends on the duration of the computation kernels.

Figure 5.2 shows speedup for all benchmarks with a shared LLC over the configuration with private LLCs. Note that the results for the 8MB shared LLC configuration are normalized to the results with the 8MB private LLCs, while the 1GB shared LLC results are normalized to the private 1GB LLCs configuration. Out of the 11 benchmarks, 7 show a speedup of over 10% with an 8MB LLC. Among those, *RBF* and *RLU* lose all the speedup with a 1GB cache. We can therefore attribute the gains to the additional cache space available to the GPU when sharing the LLC. *RBF* has a significant degree of branch and memory divergence and is largely constrained by global memory accesses [117]. Our results confirm this and show that the kernel benefits from caching due to data reuse. On the 1GB configuration the GPU is able to fit the whole working set in its private cache hierarchy; since there is no further GPU-CPU interaction after the GPU first loads the data, there is no performance benefit by sharing the LLC. We also observe a similar behavior in *RLU*.

RBP, *RPA* and *RSR* speedup is also reduced on the 1GB configuration, but still obtain 13%, 33% and 13% improvement respectively. *RSR* contains a loop in the host code calling the GPU kernels a number of iterations. After each iteration, the CPU performs a reduction with the result matrix. This data sharing pattern between GPU and CPU benefits from the faster GPU-CPU communication that a shared LLC provides. The speedup is reduced on the 1GB configuration because there is data reuse within the two GPU kernels and the larger private LLC allows more data to be kept on-chip. In *RBP* the CPU performs computations on shared data before and after the GPU kernel; the benefit of sharing the LLC is two-fold: the GPU finds the data in the shared LLC at the start of the kernel, and the CPU obtains the result faster after the kernel completes, avoiding an extra hop to the private GPU LLC.

RPA sees the largest performance improvement although there is no further GPU-CPU communication past the initial loading of data by the GPU. The gains are thus attributable to the GPU finding the data in the shared LLC at the start of the kernel. Both these benchmarks see non-negligible performance gains despite the limited GPU-CPU interaction. The reason is that the total execution time for both benchmarks is low, and the effect of the initial hits on the host-allocated data is magnified. We chose a small input set in order to run our simulations within a reasonable time-frame. On large computations this benefit would be diminished over time, hence on real hardware with larger input sets, it is likely the gains would be minimal.

RNW is the only benchmark where the performance gain of sharing the LLC actually goes up to 27% when increasing the LLC size to 1GB. This effect is due to the large input set used, with a heap usage of 512MB. The GPU's private LLC on the split configuration is not large enough to hold all the data; there is data reuse within the kernel, but due to the large working set, it is evicted out of the LLC before it is reused. On the shared configuration, the GPU benefits both from finding the data in the LLC and from being able to keep it there for reuse. In addition, after the kernel completes, the CPU reads back the result matrix, further benefiting from the faster GPU-CPU communication the shared LLC provides.

RNN experiences a small performance gain from sharing the LLC because it has GPU-CPU communication beyond the initial loading of data. When the GPU finishes computing distances, the CPU reads the final distance vector and searches for the closest value. The benchmark also benefits from the extra cache space, and thus the gains are reduced on the 1GB configuration where the 12MB input set fits in the LLC.

RGA, *RHP*, *RLA*, and *RPF* do not benefit from sharing the LLC. *RGA* launches a kernel multiple times to operate on two matrices and a vector. The benchmark benefits from caching due to data reuse, but once the matrices and vector are loaded, there is no further interaction with the CPU until the kernel completes. Although the CPU then reads the data and performs a final computation, this is just a small portion of the execution and thus the gain is negligible.

In *RLA*, the kernel is optimized to access contiguous memory locations, allowing the GPU to coalesce a large amount of memory accesses and to reduce the total memory traffic pushed into the cache hierarchy. This memory access pattern and a high data reuse translates

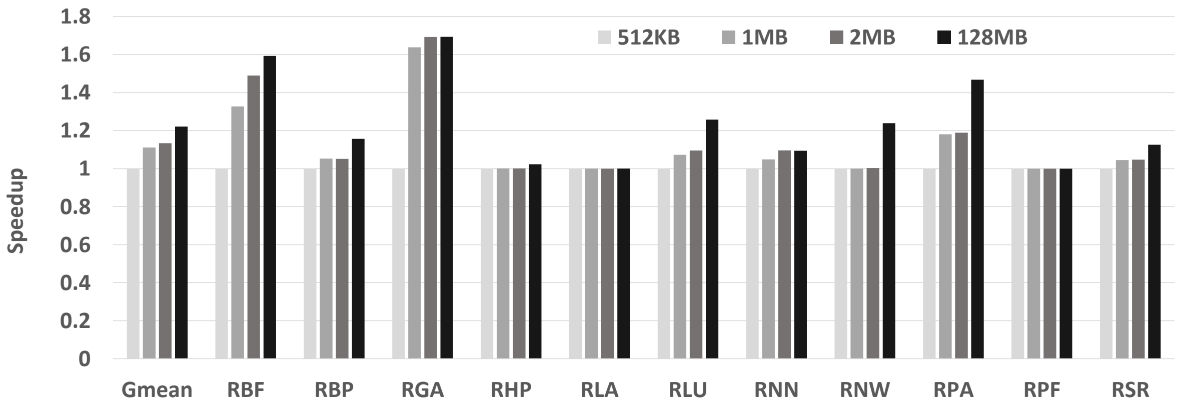


Figure 5.3: Speedup for Rodinia benchmarks as cache size increases. Each bar represents speedup for a given private GPU LLC size normalized to a configuration with a private LLC of 512KB.

into close to 99% cache hit rate in the GPU L1 caches despite an input set size of 8MB. As a consequence, sharing the LLC provides no benefit. A similar behavior can be observed in *RPF*, where the small memory footprint of the kernel allows data to fit within the GPU L1 and L2 caches.

RHP iterates multiple times operating over the same three matrices, showing data reuse with a large reuse distance. With a 1GB LLC the whole working set is able to fit in the cache, but the kernel is mostly cache insensitive and gains little from the increased hit rate. There is no GPU-CPU communication, and the small benefit of initially hitting in the LLC is diminished over the total execution time.

These results show that sharing the LLC does not provide a significant benefit for computations such as the ones found in the Rodinia benchmark suite, with minimal GPU-CPU interaction and data sharing only at kernel boundaries. The geometric mean speedup for all benchmarks is 9% on the 1GB configuration and 13% with a 8MB shared LLC, gains mostly due to the extra cache space available to the GPU. To further corroborate this hypothesis, we measure the sensitivity of the benchmarks to cache size. We run each benchmark with a split LLC configuration and with increasing private LLC sizes of 4MB, 8MB, 16MB and 1GB. Following the 1/8 ratio of GPU to CPU LLC, the GPU obtains 512KB, 1MB, 2MB, and 128MB respectively. We keep the same access latencies for all configurations in order to provide a meaningful comparison.

Figure 5.3 shows speedup as we increase cache size, normalized to the configuration with a 512KB GPU LLC. Confirming our previous findings, we see that *RBF*, *RGA*, *RLU*, *RNW* and *RPA* show sensitivity to cache size, obtaining over 20% performance increase with an large 128MB cache. *RGA* and *RBF* show very high cache sensitivity, with up to 69%

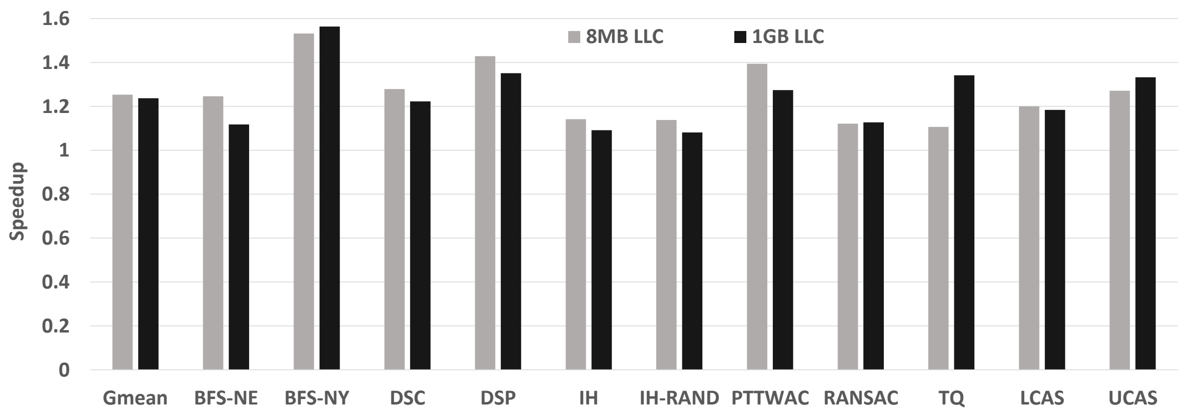


Figure 5.4: Speedup for the collaborative benchmarks with a shared LLC. Results for each cache size are normalized to the configuration with private LLCs and that same size.

and 49% improvement respectively with a more realistic 2MB cache. *RLA* and *RPF*, as discussed earlier, make almost no use of the LLC and therefore do not benefit from a larger cache. *RHP* shows data reuse and sees minor gains with a 128MB LLC, where it is able to fit the whole working set; with a smaller LLC the number of cache misses increases, but the GPU is able to hide the extra latency and the benchmark is thus mostly cache insensitive. *RBP* and *RSR* show some sensitivity to cache size, confirming the loss of speedup shown in Figure 5.2 is due to the extra cache space available. *RNN* sees some improvements up to the 2MB configuration, after which the working set fits within the 16MB of aggregated cache space. Although the 2MB of the GPU's private LLC are not enough to hold the working set, the kernel features no data reuse and therefore does not benefit from a larger LLC.

5.3.2 Collaborative Heterogeneous Benchmarks

The results presented in the previous section show traditional heterogeneous benchmarks are largely insensitive to the design of the LLC. We now analyze a set of collaborative computations with fine-grained data sharing and synchronization between GPU and CPU cores. We run the collaborative benchmarks with two CPU worker threads, with the exception of *LCAS* and *UCAS* which use only one. As in Section 5.3.1, we also run the benchmarks with an ideal 1GB LLC to isolate the gains that come from the additional cache space available on the shared configuration.

Figure 5.4 shows the speedup obtained with a shared LLC over the private LLC configuration. As with Rodinia, the results for each LLC size are normalized to the private configuration with that same size. Of the 11 benchmarks, 6 show improvements of over 20% with a shared LLC. For *BFS* we choose two different input graphs. The smaller *NY* graph has variable amount of work (and thus available parallelism) per iteration, switching often

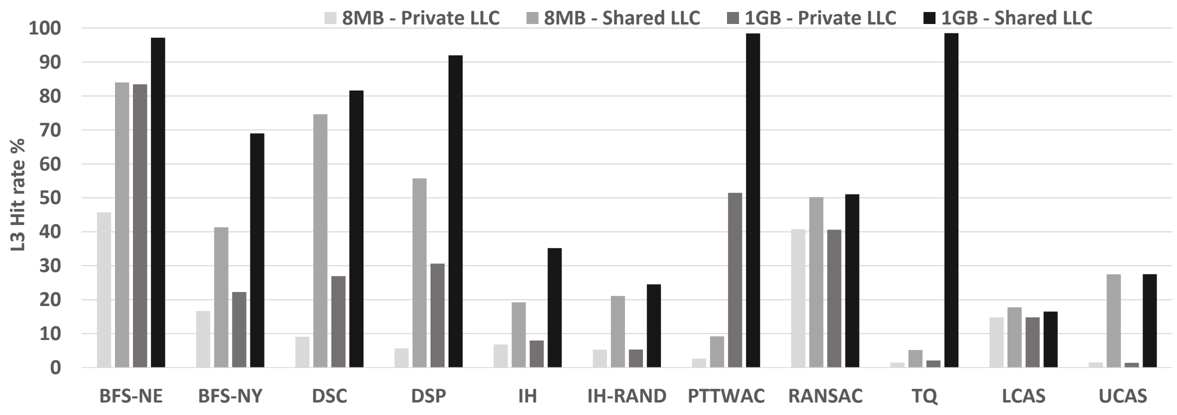


Figure 5.5: LLC hit rates for private and shared configuration.

between GPU and CPU computation. The larger *NE* graph has many iterations with a high amount of nodes, and therefore executes mostly in the GPU, switching less often between GPU and CPU. The performance gain for *BFS-NY* is higher than for *BFS-NE*, achieving as much as a 56% speedup on the 1GB configuration. This is reduced to 11% on *BFS-NE* with the 1GB LLC because with limited GPU-CPU communication, the benefit comes mostly from additional cache space.

In order to understand how sharing the LLC affects cache hit rates, we analyze L3 hit rate for both split and shared configurations. For the private LLCs configuration we calculate the aggregated L3 hit rate as explained in Section 3.3. We see in Figure 5.5 that *BFS-NE* obtains 39% higher hit rate in the private LLC configuration by increasing the size from 8MB to 1GB. As discussed, the computation is mostly performed by the GPU, where only 1MB of LLC is available on the 8MB split configuration. Being able to use the remaining 7MB that are mostly unused by the CPU provides a significant gain.

The speedup observed in *BFS-NY* supports the relevance of fast GPU-CPU communication on workloads making an extensive use of atomic synchronizations. In gem5-gpu atomics are implemented with read-modify-write (RMW) operations. A RMW instruction is composed of two parts, an initial load (LD) of the cache line with exclusive state, and a following write (WR) with the new value. Once a thread completes the LD, no other thread can read or modify that memory location until the WR finishes, guaranteeing atomicity.

Figure 5.6 shows the average access time to perform the LD part of the RMW on a shared LLC configuration, normalized to the configuration with private LLCs. We can see how *BFS* performs the operation 40% and 45% faster with a shared LLC for the *NE* and *NY* graphs respectively. On the other hand, *DSC* and *DSP* perform slower RMW LDs with a shared LLC and, nevertheless, show speedups of 27% and 42% respectively. The average time to perform the RMW LD is higher because there are more L1 misses when the exclusive LD

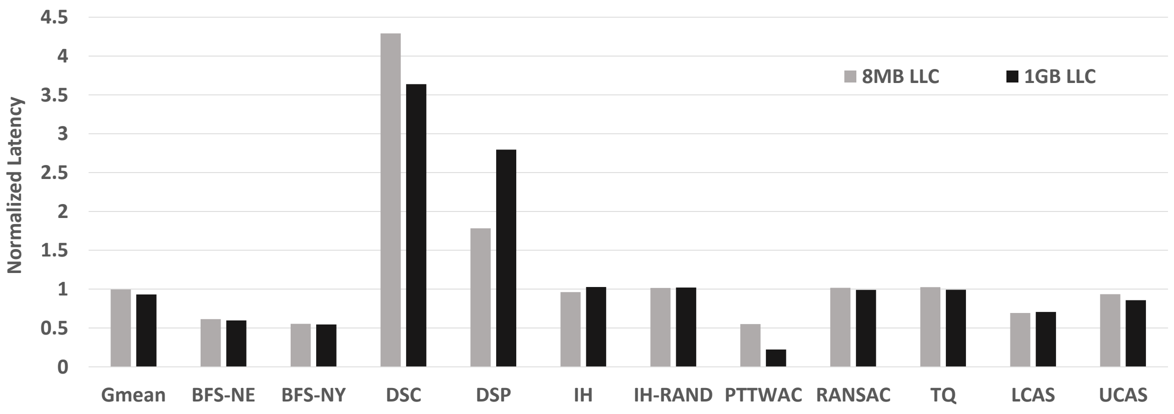


Figure 5.6: Average latency to perform a RMW LD normalized to private LLC.

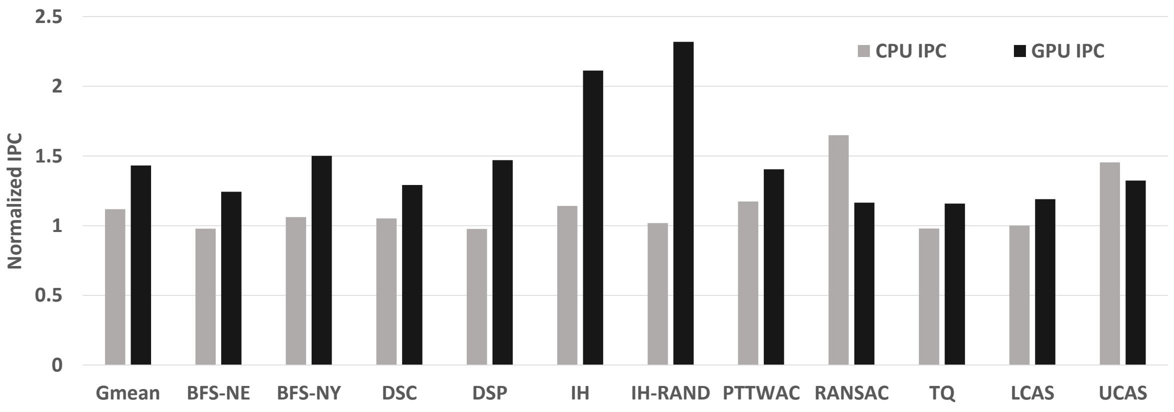


Figure 5.7: Normalized IPC with a shared 8MB LLC.

is attempted. This is a side-effect of the faster GPU-CPU communication. GPU and CPU cores compete for the cache lines holding the array of synchronization flags, invalidating each other. The shorter the latency to reach the current owner of the block, the more likely it is for a core to have relinquished ownership of the block by the time it is reused.

The reason the benchmarks still obtain a speedup with a shared LLC is that the overall memory access time for all accesses is lower. In particular for the GPU, the average latency for all the threads of a warp to complete a coalesced load instruction is 65% and 40% lower for *DSC* and *DSP* on the 8MB configuration. Figure 5.6 shows how both benchmarks go from lower than 10% LLC hit rates with a private configuration to above 80% when sharing the LLC. The benchmarks are memory bound and the reduced memory access latency caused by hitting in the shared LLC compensates the higher miss rate when performing the atomics.

Figure 5.7 shows the average GPU and CPU instructions per cycle (IPC) with a shared LLC configuration normalized to private LLCs. *DSC* and *DSP* achieve up to 30% and 47% higher GPU IPC by sharing the LLC. The more latency-sensitive CPU cores see a large increase of up to 49% on the 1GB configuration when the working set fits in the cache.

IH calculates a histogram on an input image. We configure the benchmark with 256 bins that fit within 9 cache lines (8 if aligned to block size). Our intuition was that these blocks would be highly contended and the benchmark would benefit from faster atomic operations. Interestingly we see only a relatively small speedup of 14% and 8% on the 8MB and 1GB configurations, respectively. Figure 5.6 shows that sharing the LLC does not reduce the time to perform a RMW LD. The speedup is small because in the end, the CPU is the bottleneck. The GPU benefits from multiple bins falling on the same cache line, as threads from the same warp can increment multiple bins in a fast manner. That, on the other hand, causes false sharing in the CPU caches. In addition, the work is statically partitioned, so the GPU completes its part while the CPU takes 10x longer to finish 1/8 of the image. We observe how the GPU does indeed benefit from sharing the LLC; the average latency for all the threads of a warp to finish a LD operation is reduced by 63% and 59% with a shared LLC. After the GPU finishes computing its part, the CPU remains computing, and eventually all the lines with the bins are loaded into the CPU caches, obtaining no benefit from the shared LLC. Figure 5.7 clearly depicts this. The IPC of the GPU increases over 2x on the 8MB configuration, while the CPU sees barely any improvement.

One of the consequences of using an image as the input is that we observe less conflict than expected for the cache lines holding the bins. Images usually have similar adjacent pixels, and it is likely that after obtaining a block in exclusive state to perform the atomic increment, the following pixels require incrementing a bin in the same cache line. In order to evaluate the shared LLC with a different memory access pattern, we also run the benchmark with a randomized pixel distribution (*IH-RAND*). This input reduces the number of RMW LD hits, indicating there is more contention for the lines holding the bins. Ultimately, however, the reduction in cache hits is low, as with 32 out of 256 bins per cache line it is still likely that the next atomic increment falls in a bin in the same block.

PTTWAC performs a partial matrix transposition in-place. The input matrix requires 53MB of memory, hence not fitting in the cache hierarchy on the 8MB configuration. Sharing an 8MB LLC with such a large input barely increases L3 hit rate, but still provides a 39% speedup. Figure 5.6 shows this is due to a significant reduction to the average latency required to perform the RMW LD. On the 1GB configuration the latency reduction is even larger, but the speedup is down to 27%. In this case the cache is large enough to fit the matrix, and the benchmark only benefits from faster atomics.

In *RANSAC* the CPU first performs a fitting stage on a sample of random vectors. When finished, it signals the GPU to proceed with the evaluation stage where all the outliers are calculated. This process is repeated until a convergence threshold is reached. We see that sharing the LLC improves performance by 12% for both cache sizes without speeding the atomic operations. Both GPU and CPU threads spin reading the synchronization flag when their counterpart is computing, therefore there is no contention for the block among them once it is read. The hit rate when performing the LD in a RMW is 72% for both shared and private LLC configurations, and thus the average access time is already low. The speedup in this case is not produced by faster synchronization, but from sharing the vector array. The memory footprint of the array is small, increasing L3 hit rate only by a 10%. Nevertheless, Figure 5.7 shows this 10% has a large effect on the latency-sensitive CPU, that achieves over 60% higher IPC with a shared LLC. The GPU also finds the vector array in the LLC on the first iteration, and sees a more modest 17% IPC increase. The total execution time of this benchmark is low, and thus the impact of initially finding a warm cache is magnified. As with Rodinia benchmarks, on a longer executing application the gains would diminish.

LCAS uses a CPU thread to traverse half of a linked list while the GPU threads traverse the other half, inserting in each position an identifier and atomically updating the head of the list. The cache line containing the head is highly contended, causing the atomic operations to be the bottleneck. *UCAS* traverses the list resetting the identifiers to zero. The difference between both benchmarks lies in the order the elements are accessed. Although the data structure holding the identifiers is conceptually a linked list, it is implemented as an array where the first position contains the array index of the next element. In *LCAS* the CPU inserts identifiers in consecutive array positions; GPU threads update the array position matching their thread identification number, and therefore threads from the same warp update contiguous positions.

In *UCAS* the order in which the elements are accessed is the reverse order in which the linked list was updated in *LCAS*, *i.e.* the reverse order in which the threads were able to perform the atomic operation. This difference causes the observed speedup variation. In *UCAS* the scattered access pattern causes many blocks to be moved back and forth between GPU and CPU, and is reflected by the near 0% L3 hit rate seen in Figure 5.5. In this case the data migration latency from GPU to CPU is also an important factor. The results show that although *UCAS* achieves on average a lower latency reduction to perform a RMW LD with a shared LLC, the benefit of faster data movement actually results in a higher speedup compared to *LCAS*.

In *TQ* the CPU is in charge of inserting 128 frames in several queues. GPU blocks dequeue individual frames and generate their histogram. As the histogram of each frame is only accessed by one GPU block, it will be kept in the L1 cache ensuring low latency for RMW operations already on the private LLC configuration. Additionally, the number of atomics on the control variables of the queues is very small compared to that of the atomic operations on the histograms. Thus, the average latency to perform the RMW is not reduced by sharing

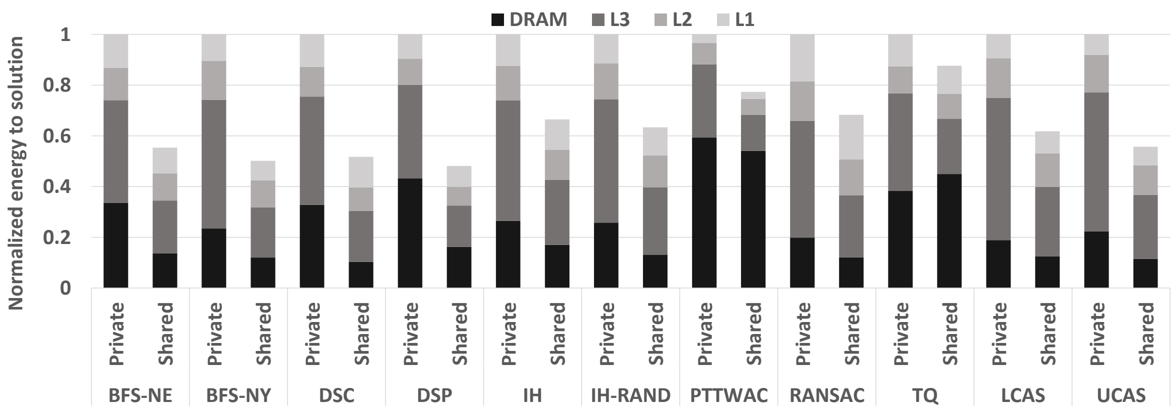


Figure 5.8: Energy-to-solution normalized to the configuration with private LLCs.

the LLC. However, the LLC hit rate is higher on the shared configuration, because the GPU blocks will eventually read frames previously cached by the CPU thread when enqueuing them. This explains the 10% speedup on the 8MB configuration. The improvement is much higher on the 1GB configuration (34%) because the larger cache can keep the entire pool of 128 frames (54MB) and the queues.

5.3.3 Energy

We provide an analysis of the energy implications of sharing the LLC when executing collaborative heterogeneous applications. Figure 5.8 shows energy-to-solution with an 8MB LLC normalized to the configuration with private LLCs. We show a breakdown of the energy consumption of the different components of the memory subsystem, the different cache levels and off-chip DRAM.

Our results show that sharing the LLC decreases energy-to-solution on all benchmarks by at least 20%. *BFS*, *DSC*, *DSP* and *UCAS* see a reduction of over 40% while *IH*, *RANSAC* and *LCAS* consume over 30% less energy compared to a configuration with no shared LLC. Static energy is reduced on all benchmarks due to shorter execution times. The other major reduction comes from lower L3 dynamic power. A shared LLC increases hit rates and avoids the extra requests and coherence traffic caused by a cache miss. This can lead to significant energy savings, as in *BFS*, *DSC*, *RANSAC*, *LCAS* and *UCAS* with 2.56x, 2.11x, 1.64x, 2x and 2.17x lower L3 energy consumption respectively.

The third reduction in energy-to-solution comes from DRAM dynamic power. The results we present in this section are of the benchmark's region of interest; data has already been allocated and initialized. In most cases the data is already on-chip, and therefore the total number of off-chip accesses is already low. Sharing the LLC improves resource utilization by allowing GPU and CPU cores to access the full cache space available, and allows for data to stay longer in the hierarchy, further reducing off-chip traffic. The exceptions are *PTTWAC* and *TQ*. Both benchmarks have a working set size far larger than the cache hierarchy, and must still load data from DRAM. The shared LLC minimally reduces off-chip accesses in *PTTWAC*, and slightly increases it in *TQ*. The reason is that the frames sometimes evict the queues from the shared LLC, causing off-chip write-backs and subsequent reloads, while on the private configuration the queues are able to stay in the CPU's LLC.

5.4 Summary and Concluding Remarks

The work presented in this chapter is motivated by the lack of efforts focusing on the effects of resource sharing when executing collaborative heterogeneous computations. We believe the tighter integration of CPU cores with GPUs and other accelerators will change the way we understand heterogeneous computing in the same way the advent of multicore processors changed how we think about algorithms. In order to understand the impact of sharing the last-level cache on an integrated heterogeneous architecture, we perform an evaluation of two different cache hierarchy designs on a set of heterogeneous benchmarks.

First, we perform an evaluation of the popular Rodinia benchmark suite modified to leverage the unified memory address space. We find such GPGPU workloads to be mostly insensitive to changes in the cache hierarchy due to the limited interaction and data sharing between GPU and CPU. We then evaluate a set of collaborative heterogeneous benchmarks specifically designed to take advantage of the fine-grained data sharing and low-overhead synchronization between GPU and CPU cores that integrated architectures enable. We show how these algorithms are more sensitive to the design of the cache hierarchy.

Our results indicate that sharing the LLC in an integrated GPU-CPU system is desirable for heterogeneous collaborative computations. The first benefit we observed is due to the faster synchronization between GPU and CPU; in applications where fine-grained synchronization via atomic operations is used and many actors contend to perform the atomics, accelerating this operation provides considerable speedups. The second benefit is due to data sharing; if GPU and CPU operate on shared data structures, sharing the LLC will often reduce average memory access times and dynamic power consumption. We have observed this effect both with read-only and private read-write data.

The third benefit we observed is due to better utilization of on-chip resources; a cache hierarchy where the LLC is partitioned will often underutilize the available cache space, while sharing it guarantees full utilization if needed by the application. This insight is specially relevant since it applies to any kind of computation, not only collaborative. A split LLC configuration executing GPU-only or CPU-only code will not utilize a portion of the LLC, wasting resources and likely power.

Yet, resource sharing between such disparate computing devices introduces new challenges. We have seen an increase of conflict misses specially with large input sets. In the benchmarks we evaluated the benefits of sharing the LLC offsets the drawbacks. However, we are only focusing on computations that fully leverage the characteristics of integrated heterogeneous architectures. In the last few years researchers have shown and proposed solutions for the challenges of resource sharing with other types of workloads, and further investigation is required if the trend of GPU-CPU integration is to continue.

Overall, our results show that Rodinia benchmarks with coarse-grained GPU-CPU communication experience an average 13% speedup using an 8MB shared LLC versus a private LLC configuration, mainly due to the extra cache space available to the GPU or short execution times. Collaborative computations that leverage the shared virtual address space and fine-grained synchronization achieve an average speedup of 25% and of up to 53% with an 8MB shared LLC. In addition, energy-to-solution is reduced for all benchmarks, with 9 of the 11 collaborative benchmarks evaluated showing reductions of more than 30% compared to the configuration with private LLCs. The energy savings come mostly from lower static power consumption due to shorter execution times and reduced L3 and DRAM dynamic power consumption.

Summarizing, the benefits we have listed encourage a rethinking of heterogeneous computing. In an integrated heterogeneous system, computation can be divided into steps; each step can be executed on the computing device that is best suited for, seamlessly sharing data structures among computing elements and synchronizing via fine-grained atomic operations. Sharing on-chip resources such as the last-level cache can provide performance gains if the algorithms fully leverage the capabilities of these integrated systems, and will guarantee a better utilization of available cache space.

Chapter 6

Efficient Data Sharing on Heterogeneous Architectures

6.1 Motivation

As discussed in Chapter 5, the current trend for heterogeneous architectures is towards tighter coupling of GPU and CPU. Yet, while physical integration of the GPU on-die with the CPU cores is becoming the norm, the majority of heterogeneous systems used nowadays in the field of HPC still use discrete GPUs connected to a multicore machine through an interconnect such as PCIe or NVLink. Discrete devices, implemented on a separate, (relatively) large chip with billions of transistors, usually contain higher core counts than integrated GPUs and use specialized graphics memory that provides higher bandwidth than commodity DRAM. Hence, the computing potential of discrete GPUs currently dwarfs that of integrated systems.

Programmability is one of the main challenges of discrete heterogeneous architectures [118]. Manually managing two different memory pools and efficiently copying data back and forth between them is a time-consuming and error-prone endeavor. Over the years GPGPU has become more accessible due to the introduction of shared virtual memory and automatic data movement. Today, the Pascal line of GPUs by NVIDIA is able to perform on-demand paging of memory to the GPU transparently to the user [119]. This feature, possible due to the support for GPU-initiated page faults simplifies heterogeneous programming and allows discrete GPUs to execute collaborative computation and to use complex, pointer-based data structures such as binary trees and linked lists.

Unfortunately, relying on the CUDA runtime to manage data movement comes at a price, and that is performance. Automatic memory management is convenient but suffers from many drawbacks, preventing heterogeneous systems from achieving their full potential. Demand paging in GPUs introduces significant overheads because GPUs are not yet able to execute their own page fault handling routines and must forward them to the CPU. In traditional heterogeneous applications, input data is initialized in the host and copied to the device to take advantage of the local high-bandwidth memory; after the computation, the results are copied back to the host. In this model, data is copied only once in each direction. In collaborative heterogeneous applications, on the other hand, host and device operate on shared data structures and data may migrate many times in both directions. In such computations demand paging is even more taxing because the page fault latency must be paid on every migration.

In this chapter we analyze the inefficiencies of the current demand paging scheme found in discrete GPUs. We argue that migrating full OS-defined memory pages on every memory access is inefficient, as fine-grained data sharing between GPU and CPU causes unnecessary data transfers. Furthermore, if both host and device operate on memory within the same physical page, a ping-pong or *false sharing* effect may occur, severely degrading performance of both CPU and GPU.

To solve these problems, we propose a memory organization and dynamic migration scheme to efficiently share data between host and device. Our goal is to enable heterogeneous systems with discrete GPUs to efficiently execute collaborative computations. In our scheme, only the first GPU access to a memory page incurs a long-latency page fault, significantly improving the performance of computations where data is migrated back and forth between host and device. We leverage the observation that heterogeneous applications rarely need to modify the page table of the heterogeneous process. Therefore, copying the corresponding page table entry on the first GPU access is sufficient to perform virtual address translation in the GPU for all subsequent accesses to that page.

In addition, the memory organization we propose, based on previous work on DRAM caches, reduces the granularity of migrations from full pages to cache lines. The advantages are two-fold: first, moving away from OS-defined memory pages avoids expensive page table manipulations and allows for hardware-managed migration of data transparently to the user and OS. Second, we save bandwidth and reduce false sharing by migrating only the cache lines that are demanded and not surrounding memory regions that may be in use elsewhere.

6.2 Demand Paging on GPUs

Resolving GPU-initiated page faults is an expensive operation that requires: forwarding the fault to the host, interrupting a core to execute a privileged page fault handling routine, manipulating GPU and CPU page tables, sending TLB shootdowns and setting up the GPU's DMA engine to migrate the page. The most common interconnect used in heterogeneous systems is PCIe, with an approximated round-trip time (RTT) of 2 μs [120]. Handling a fault

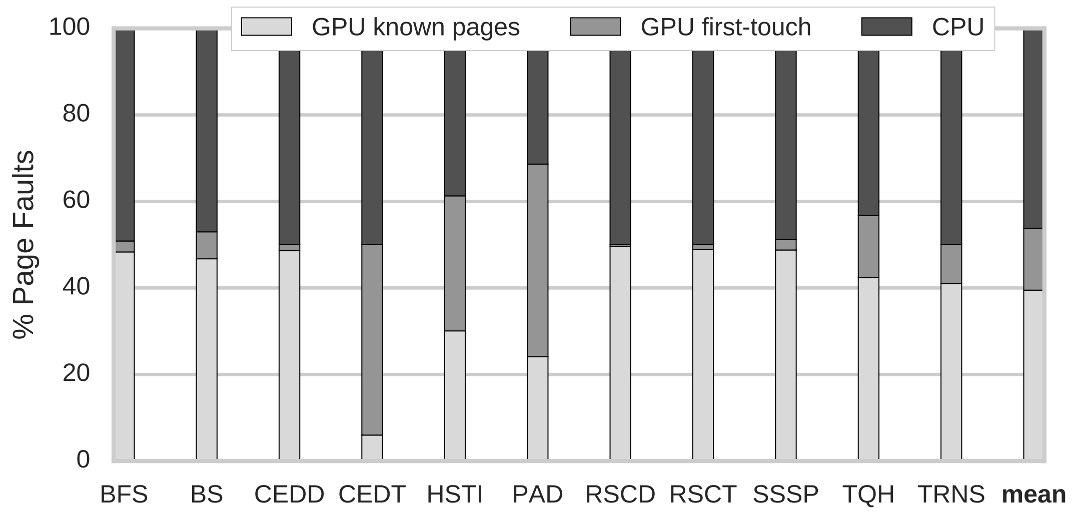


Figure 6.1: Breakdown of all page faults caused by demand paging.

requires multiple messages between GPU and CPU, and thus resolving a GPU-initiated page fault can take anywhere between 20 and 50 μs [70].

Recent work in the literature proposes hiding this latency by leveraging the highly-threaded nature of GPUs and by prefetching memory pages [70]. While this is a sensible approach for traditional heterogeneous applications where the GPU reads large regions of contiguous memory and data stays in the GPU during kernel execution, collaborative heterogeneous computations display a different behavior. We will show that the current scheme of demand paging is particularly inefficient on these computations because data is shared at fine granularities and migrated multiple times between host and device, incurring the full page fault latency every time.

6.2.1 Page Faulting on Known-Pages

Figure 6.1 shows a breakdown of all the page faults raised on a system with demand paging during the execution of a set of collaborative heterogeneous benchmarks from Chai [109]. Details about the benchmarks can be found in Section 3.2.2 and about the simulation infrastructure used in Section 3.1. *GPU first-touch* represents faults caused by the first GPU access to a page allocated in the CPU; *GPU known pages* are faults caused by GPU accesses to pages that were migrated at some point to the GPU but are now in CPU memory; *CPU* are faults caused by CPU accesses to pages located in GPU memory. As discussed in Section 3.2.2, we only evaluate the region of interest of every benchmark, skipping initialization and clean-up phases. We therefore do not consider the CPU-initiated page faults caused by the operating system's lazy-allocation, *i.e.* faults caused on the initialization of input data.

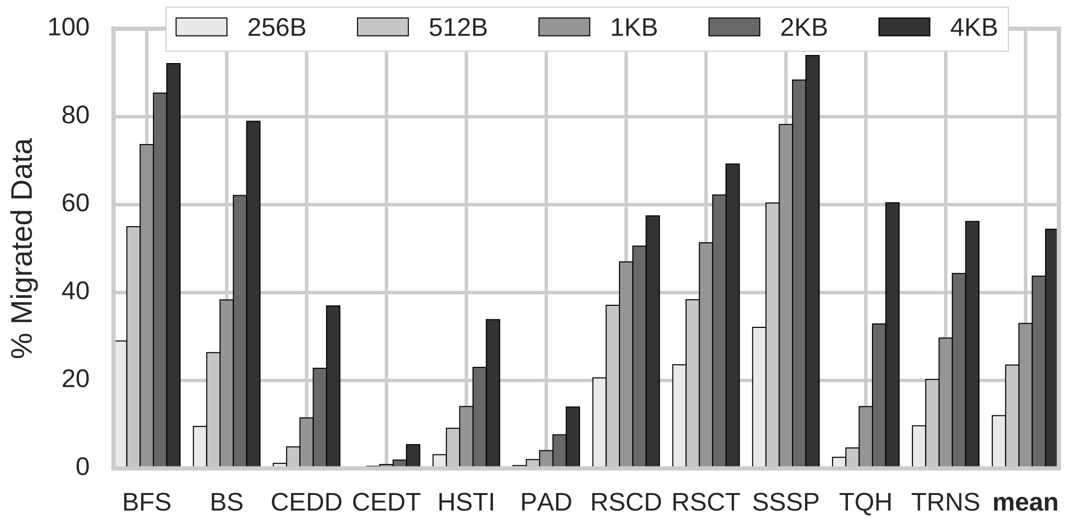


Figure 6.2: Percentage of unused data with different migration granularities.

In the figure we can see how a large percentage of GPU-initiated faults are caused by *known pages* that have been migrated to the GPU and back to the CPU at least once. In benchmarks such as *BFS*, *CEDD*, *RSCD*, *RSCT* and *SSSP* only a small number of memory pages are referenced and migrated multiple times back and forth between the two memories. On average, 74% of GPU-initiated page faults and 39% of all the page faults are caused by *known pages* migrating to the GPU, and 42% of all the faults are caused by migrations back to the *CPU*. The goal of the work presented in this chapter is to reduce the latency of migrating *known pages* to the GPU and back to the CPU.

6.2.2 Unused Data and False Sharing

The current demand paging scheme can also be inefficient because full OS-defined memory pages are migrated on every memory access. Traditional GPU applications stream through large contiguous memory regions and are likely to reference entire pages, but warp memory divergence and the use of irregular data structures can result in a more irregular memory access pattern. In addition, collaborative applications share data at a finer granularity; copying a 4KB memory page on every access can waste bandwidth by migrating unneeded memory.

Figure 6.2 shows the percentage of unnecessarily migrated cache lines as we increase the granularity of migrations. We consider a cache lines as unused if it is migrated back and forth from one memory to the other without being referenced. Cache line size is 128 bytes in our simulated architecture; we show results for migration sizes going from two cache lines to a full page (4KB typically in Linux consumer systems). We can see how on average 57% of all the migrated cache lines are transferred unnecessarily at least once when migrating full pages. That number is reduced to 14% when only two cache lines are migrated.

In three benchmarks, *BFS*, *BS* and *SSSP*, more than 75% of all the copied data is unnecessarily migrated at least once. In addition, if GPU and CPU concurrently reference memory within the same page, the page will suffer from a ping-pong or *false sharing* effect. False sharing is a well known problem in shared memory multiprocessors [121, 122] caused by two or more cores simultaneously accessing different bytes within the same line. Due to the cache line granularity the cache subsystem works at, the line is migrated back and forth between the cores' private caches, degrading performance. In Section 6.5 we provide a detailed analysis of how collaborative benchmarks are affected by false sharing with page-sized migrations.

6.3 Efficient Data Sharing in Heterogeneous Architectures

This section describes the main design points and implementation details that enable efficient data sharing in heterogeneous systems. We first describe the memory organization that allows reducing the granularity of migrations to cache lines, as well as modifying their physical address transparently to the OS. We then show how this reduces the migration latency of data that has been previously copied to the GPU. Finally we explore the idea of grouping multiple data migrations to amortize DMA setup times and interconnect latency.

6.3.1 Heterogeneous Memory Organization

The goal of the work presented in this chapter is to efficiently migrate data between two different memory pools transparently to the user. Our first concern is to reduce the granularity at which data is migrated, as we have shown how migrating full pages unnecessarily transfers data not demanded, wasting bandwidth and potentially causing false sharing. In addition, we require a scheme that migrates data without involving the CUDA driver or the operating system as much as possible, as doing so introduces overheads and long latencies that are to be avoid.

DRAM caches have been previously proposed for heterogeneous memory organizations, where two pools of memory with different characteristics are combined and movement of data between them must be handled transparently to the user to maximize performance. In particular, we base the design of our memory organization on CAMEO [82]. CAMEO fulfills both requirements for our efficient data migration scheme: it performs data movement between two DRAM memories of different technologies at cache line granularity, and it does so transparently to the user and OS, without page table manipulations. In addition, as opposed to similar work on DRAM caches, CAMEO maintains the two memories in the memory space visible to the OS, allowing the full aggregate memory range to be addressable by the applications.

CAMEO was proposed for a heterogeneous memory system with *vertical* integration, where a 3D die-stacked DRAM is integrated on-chip between the last-level cache and off-chip memory. In such an architecture, the stacked memory is always accessed first, and only on a miss an access to off-chip memory is required. Our heterogeneous architecture, on the other hand, contains a memory organization with *horizontal* integration, where both memories can be accessed first by either the GPU or the CPU. While CAMEO can store in the stacked DRAM the metadata required to locate every cache line in the system, our design requires duplicating the metadata and keeping it coherent.

The memory organization we propose in this chapter divides the physical memory space into *Congruence Groups*, with the total number of groups N being equal to the number of lines in GPU memory. The set of lines that can map to a given location in GPU memory forms one Congruence Group. The Congruence Group for a line is identified by the bottom $log_2(N)$ bits of the physical line address. On a system with a 3 to 1 ratio of CPU to GPU memory, a Congruence Group is composed of four cache lines. For simplicity, we assume the addressable space starts from GPU memory and CPU memory continues afterwards. Figure 6.3 shows an example of the memory organization, where four lines A, B, C and D form a Congruence Group.

When a line is migrated from one memory to the other, it is swapped with another line from the same Congruence Group. A structure called the *Line Location Table* (LLT) is used to identify the location of every line in memory. A *Location Table Entry* (LTE) contains the real physical location of all the lines in the Congruence Group, and is updated whenever there is a line swap.

Figure 6.4 shows how the LTE for a Congruence Group is updated as lines migrate to the GPU and are swapped with lines from the same group. Initially, all lines are in their starting physical location. When the GPU requests line C, currently in CPU memory, a swap operation is done, migrating C to the GPU and A to the physical location where C previously resided. As the execution continues, lines are swapped as they are requested by the GPU, updating the LTE.

On the system described earlier with a 3 to 1 ratio of CPU to GPU memory, each LTE is a four-entry tuple with two bits per line identifying in which of the four possible physical locations within the congruence group a line is located. The storage requirements for the LLT on a system with tens of gigabytes of memory are therefore non-negligible. In this example where each LTE needs only 8 bits, a 32GB system with 128B cache lines would need 64MB

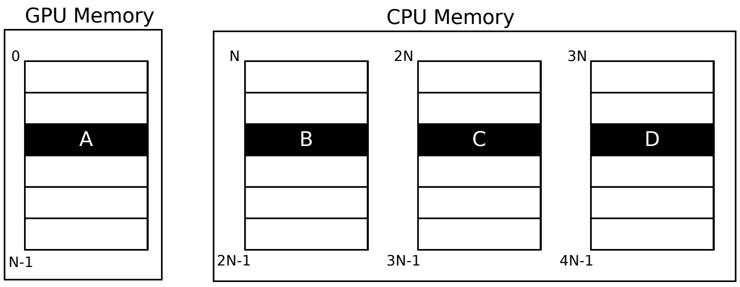


Figure 6.3: Lines A, B, C and D form a Congruence Group. As lines are migrated to the GPU, they are swapped with other lines from the same group.

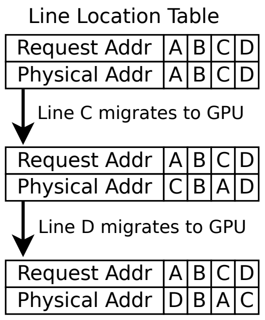


Figure 6.4: Line Location Table updates as lines are migrated to GPU memory.

of storage (32GB divided by 512B, the size of each congruence group). Since the storage requirements are too high to realistically place the LLT in on-die SRAM memory, it is kept in off-chip DRAM memory.

In order to minimize access time, the LLT is co-located with the data itself in DRAM memory. By doing so, we avoid the need for two memory accesses on every memory request, one to read the LLT and find the real location, and another to read the data. The LTE metadata for every cache line is therefore appended to the line itself in the DRAM row buffer. In this manner, a single burst read of DRAM[1] will provide the physical location, and if the line is found in that location, the data itself. Only if the LTE identifies the line as located elsewhere, another access is then required.

In the architecture we evaluate in this work, DRAM row buffer size equals 2KB. In a system with 128 byte-sized cache lines, each row buffer holds 16 cache lines (2KB / 128B = 16). In order to co-locate the LLT with the data in DRAM memory, we need to sacrifice some DRAM space. Using the 128 bytes of space of one cache line is more than sufficient to hold all the LTEs for lines in a row buffer, leaving us with 15 cache lines per row buffer and a DRAM utilization of 93.75% (15/16). A loss of 6.25% of DRAM memory is deemed as an acceptable trade-off on systems with tens of gigabytes of memory, and it could also be reduced by using smaller cache lines.

For simplicity, we remove the last 64MB from the OS-addressable space on both GPU and CPU memories to allocate the LLT. The shift of data in memory caused by appending the LTE to every cache line needs to be adjusted for before accessing the DRAM, in the following manner: *LineAddrX = (X + X/15) - LinesIn64MB*, where *X* is the address of line requested. The original CAMEO paper [82] suggests using residue arithmetic to perform the

[1]DRAM burst size must be large enough to read both the data and the LTE metadata with one access. See Table 6.1 for details on the configuration of the simulated architecture.

division by a constant with only a few adders. This operation is done in parallel with the last-level cache lookup in order to hide the latency.

On a system with *vertical integration* where the DRAM cache is always accessed first, it is sensible to store the LLT in the DRAM cache. In a heterogeneous system with *horizontal integration*, placing the LLT in CPU (GPU) memory would require the GPU (CPU) to read it through the high-latency interconnect even if the line was actually in the GPU (CPU). We therefore need to replicate the LLT in both host and device memories, with the additional complexity of keeping them coherent. Fortunately, since LTEs are only modified on a line migration, we can use the DMA engine to serialize operations and ensure that both copies of the LLT are always kept coherent.

6.3.2 Avoiding Host Intervention

Since current Pascal-based GPUs are not yet able to execute fault handling routines, the current demand paging scheme requires sending the fault to the host to be processed. Involving the host on every GPU-initiated page fault introduces significant overheads, as it requires sending several messages through a high-latency interconnect, interrupting a CPU core to execute a privileged page fault handling routine and updating both GPU and CPU page tables. The goal of our memory organization and migration scheme is to simplify the handling of GPU-initiated page faults, avoiding host intervention as much as possible.

The memory organization explained in Section 6.3.1 can migrate data to and from GPU memory, hence modifying the physical address, without invalidating the existing virtual to physical address mapping. This allows us to avoid updating GPU and CPU page tables on every migration. Still, the first GPU access to a memory page allocated in the CPU will not find the corresponding page table entry (PTE) in the GPU's page table. It is therefore necessary to update at least once the GPU's local page table in order to provide virtual to physical address translation.

In the proposed scheme, the first GPU access to a memory page generates a long-latency page fault similar to current Pascal-based GPUs. After resolving the fault, the PTE referenced is copied to the GPU's page table. We leverage the observation that the page table of the heterogeneous process is rarely modified during runtime[2], and it is therefore sufficient to copy the PTE on the GPU's first access to perform physical to virtual address translation in the GPU. Subsequent GPU accesses to the same page are able to find the mapping either in the local TLBs or the local page table, and do not incur a long-latency page fault.

[2]This observation is based on our experimental evaluation, where none of the benchmarks evaluated ever required modifying the page table of the heterogeneous process from kernel launch to kernel completion.

6.3.3 Efficient Fine-Grained Migration

Although the overheads of migrating data on-demand are significantly lower with our proposed scheme compared to the baseline architecture, the latency of the PCIe interconnect adds non-negligible delays to every data migration. In order to amortize the cost, it is necessary to transfer large blocks of data that maximize the utilization of the interconnect.

GPU applications typically display a bursty memory access pattern where multiple memory accesses are issued from different warps in a short span of time. We leverage this behavior by grouping multiple data migration requests and processing them in batches. Due to the data granularity we work with, a batch of data migrations will contain multiple cache line requests from different warps to potentially non-contiguous memory. The DMA engines found in current NVIDIA GPUs do not allow for data transfers of disjoint memory regions, and would require multiple DMA commands to process all migrations. In order to efficiently copy data at smaller granularities, the DMA engines must support scatter/gather operations.

DMA engines with support for scatter/gather are already available in accelerators such as FPGAs [123], in some ARM chips [124] and in the Cell chip [125]. As heterogeneous computations become more collaborative with fine-grained data sharing between host and device, we believe allowing DMA transfers of disjoint memory regions will bring significant performance gains, and therefore we advocate for supporting them in future GPUs.

Migration requests are aggregated in a small buffer or staging area (SA) and sent in batches. After a defined time interval or when the SA is full, whichever comes first, all buffered migrations are grouped together to create a single DMA list command. A DMA list command is an array of source/destination addresses and lengths. In order to synchronize the swapping operation and avoid overwriting data, we use the SA of the device initiating the migration to store a copy of the lines to be swapped out. The process of swapping lines involves two DMA operations. First, data from the remote (non-initiator) memory is copied to the local (initiator) memory; then, the backed-up copy of the data is transferred to the remote memory.

Once the initiator receives the data, pending memory accesses can complete and need not wait for the other DMA to finish. During a swap operation the SA can continue receiving migration requests. To ensure the backed-up data is not overwritten until the DMA has completed, we divide the SA in two: one half holds the data for the current migration and the other half contains the data for the next migration. Once a migration fully completes and data has been swapped, the current half can be cleared.

Two physical registers are used to keep the starting address of current and future SAs. When the SA is full or a time interval concludes, the value of the registers is updated and

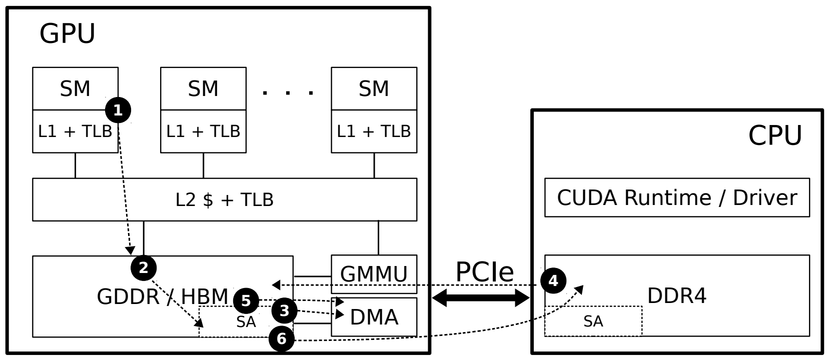


Figure 6.5: High level overview of the architecture and steps followed on a GPU-initiated migration.

a new DMA list command is generated with the buffered migration requests. We evaluated multiple SA sizes and time intervals and found 4MB and 10μs to be sufficient to hold the migration requests generated in all benchmarks. In addition to the 64MB removed from the addressable memory space discussed in Section 6.3.1, we remove 4MB of both GPU and CPU memories to be used as staging areas.

The use of the SA to buffer migration requests may break the atomicity of line swaps if two or more lines from the same congruence group need to be migrated simultaneously. The first of a set of conflicting migrations that takes place will modify the location of lines from the group and the corresponding LTE. Following migrations for the same congruence group that have been backed up into the SA will attempt to copy the wrong lines, resulting in an inconsistent memory state. In order to detect such situations and maintain consistency, the LTE of each line migrated is compared to the LTE of the location it is copied into. An LTE mismatch signifies a previous migration modified the location of lines from that group, in which case we do not perform the copy. After the swap operation completes, all load instructions will be retried, and thus a new migration can be started with the updated LTE.

Figure 6.5 shows a high level overview of the architecture and the steps followed on a GPU-initiated migration, as described next (for simplicity we assume the data migrated has already been copied to the GPU at some point, and therefore does not cause a page fault).

1 An SM executes a load instruction; GPU memory is read and the LTE provides the current location of the cache line in CPU memory.

2 The cache line located in GPU memory from the same congruence group and the corresponding LTE are copied to the staging area; the address requested by the SM is added to the destination vector and the address obtained from the LTE to the source vector

3 After a time interval or when the staging area is full, a DMA list command is generated from the source and destination vectors and inserted in the DMA command queue.

4 The DMA operation copies data from CPU to GPU memory; the LTEs of all cache lines swapped are updated in GPU memory; the warp instructions pending data migrations can now proceed.

5 A new DMA list command is generated with the GPU SA addresses in the source vector and the previous source vector addresses (in CPU memory) in the destination vector; the command is inserted in the command queue and the DMA transfer initiated.

6 Data from the SA is copied to CPU memory and the LTEs are updated; the swapping operation completes.

Similarly, the CPU is also able to initiate migration operations in the other direction. The main difference is that it must write the DMA command over the interconnect into the GPU's DMA command queue, with the additional latency it entails.

This procedure depicts the steps followed when data from *known-pages* is migrated back to the GPU. Alternatively, when the GPU first accesses a virtual address for which no translation is yet available, a long-latency page fault is generated. These faults are forwarded to the CUDA runtime running on the host where they are enqueued to be processed in batches. Once all the faults are resolved and the physical addresses are known, the runtime initiates a migration operation for the lines requested. First, the runtime starts or waits for completion of the current CPU-initiated migration using the current half of the SA; then, CPU memory is accessed for every physical address translated.

If the line matches its initial position in the congruence group, it is directly copied to the CPU SA; if the LTE indicates a different address, an additional access is needed to the correct location. A DMA list command is generated to copy data from device to host; the source vector is populated with the congruence groups' GPU addresses and the destination vector with the addresses of the lines in CPU memory backed-up in the SA. The DMA command is inserted into the GPU DMA command queue and the data is transferred.

Then, a new DMA command is executed with the CPU's SA addresses as source and previous source as destination; data is copied from host to device and the migration completes. It should be noted that this operation increases the latency of GPU-initiated page faults compared to the baseline system, as it requires an additional DMA operation to swap data between host and device. Nevertheless, this operation is only necessary on the first GPU access to a page.

Table 6.1: Simulation Parameters for the Discrete Heterogeneous Architecture.

CPU	
Cores	10 @ 2 Ghz
L1D Cache	64KB - 2 way - 1ns lat.
L1I Cache	32KB - 4 way - 1ns lat.
L2 Cache	2MB - 8 way - 8ns lat.
GPU	
SMs	16 - 32 lanes per SM @ 1.4 Ghz
L1 Cache	16KB + 48KB shared mem. - 4 way - 22ns lat.
L2 Cache	1MB - 16 way - 4 slices - 63ns lat.
CPU Memory	
DDR4	24GB - 4 channels - 2 ranks - 16 banks @ 1200 MHz
Burst length / Row size	8 / 2KB
RAS/RCD/CL/RP	32 / 14.16 / 14.16 / 14.16 ns
RRD/CCD/WR/WTR	4.9 / 5 / 15 / 5 ns
GPU Memory	
GDDR5	8GB - 4 channels - 1 rank - 16 banks @ 1000 MHz
Burst length / Row size	8 / 2 KB
RAS/RCD/CL/RP	28 / 12 / 12 / 12 ns
RRD/CCD/WR/WTR	6 / 3 / 12 / 5 ns

6.4 Methodology

In order to evaluate the memory organization and migration scheme we propose in this chapter we analyze its impact when running the set of collaborative benchmarks from the Chai benchmark suite described in Section 3.2. We run all benchmarks several times with increasing migration granularities. For each granularity, the block of data transferred is aligned to the migration size, *i.e.*, with 4KB migrations, the physical page where the memory access falls into is migrated; with 2KB migrations, the upper or lower half of the page, *etc.*

We simulate a heterogeneous system composed of a 10 core CPU and a GPU with 16 Maxwell-like SMs connected through a PCIe 3.0 interconnect. The PCIe link has a $2\mu s$ RTT [120] and 16 GB/s of bandwidth. In Section 6.5.3 we explore the effect of varying the RTT of the interconnect. Unless stated otherwise, we run all benchmarks with 8 CPU worker threads. Table 6.1 lists the configuration parameters of the system. L1 data and instruction caches and the L2 cache are private for each CPU core. L1 caches are private for each SM while the L2 is shared among all SMs. Cache line size is 128 bytes across the whole system. The CPU has 24GB of DDR4 memory and the GPU features 8GB of GDDR5.

Unless stated otherwise, all the results presented are normalized to the baseline configuration. The baseline architecture behaves like current GPUs with support for demand paging.

On a GPU access to a page located in CPU memory, a page fault is generated and forwarded to the CUDA driver running on the host to be handled; the driver then raises a software interrupt for a CPU thread to execute a privileged page fault handling routine. After the fault is serviced, the faulting page is copied to the GPU and both GPU and CPU page tables are updated. Subsequent CPU accesses to the page cause a migration back to CPU memory that invalidates the GPU's page table entry, thus incurring a new fault if the page is referenced again by the GPU. Since we are simulating a full-fledged system running a Linux kernel, the time to resolve a page fault is non-deterministic and depends on factors such as the state of the interrupted core, whether the entry is cached, swapped out to disk, *etc.*

6.5 Experimental Evaluation

This section presents an evaluation of our proposed memory organization and dynamic migration scheme. We analyze our scheme with various migration granularities and identify those benchmarks that suffer from false sharing when large migration sizes are used. In addition, we provide an analysis of how decreasing the link latency affects the feasibility of fine-grained migrations.

6.5.1 Migration Granularity

We measure the execution time for all benchmarks as we increase the granularity of migrations from 128 bytes corresponding to one cache line up to a full 4KB page.

Figure 6.6 shows execution time for Chai benchmarks normalized to the baseline system implementing demand paging. We see how our scheme with cache line-size migrations is able to reduce execution time by 15% on average for all benchmarks, although severely degrading performance on *CEDD*, *CEDT*, *PAD* and *TRNS*. As we have shown, 4KB migrations inefficiently migrate data that is not needed, yet with our scheme they provide a significant 47% execution time reduction on average over the baseline. Overall, 2KB migrations provide the best results with a 50% execution time reduction on average for all benchmarks. *BS*, *HSTI* and *TQH* obtain the best performance with 128 byte-sized migrations, while *BFS*, *RSCD* and *SSSP* see a significant speedup with our scheme in all configurations.

CEDD and *CEDT* are two implementations of an imaging algorithm that analyzes frames of a video. Small migrations degrade performance because 650 bytes of memory are read per frame, and performing many small migrations to copy the data is inefficient. We see the performance improving significantly for *CEDD* once the migration size increases to 1KB.

CEDD implements a data partitioning scheme while *CEDT* partitions the work by tasks.

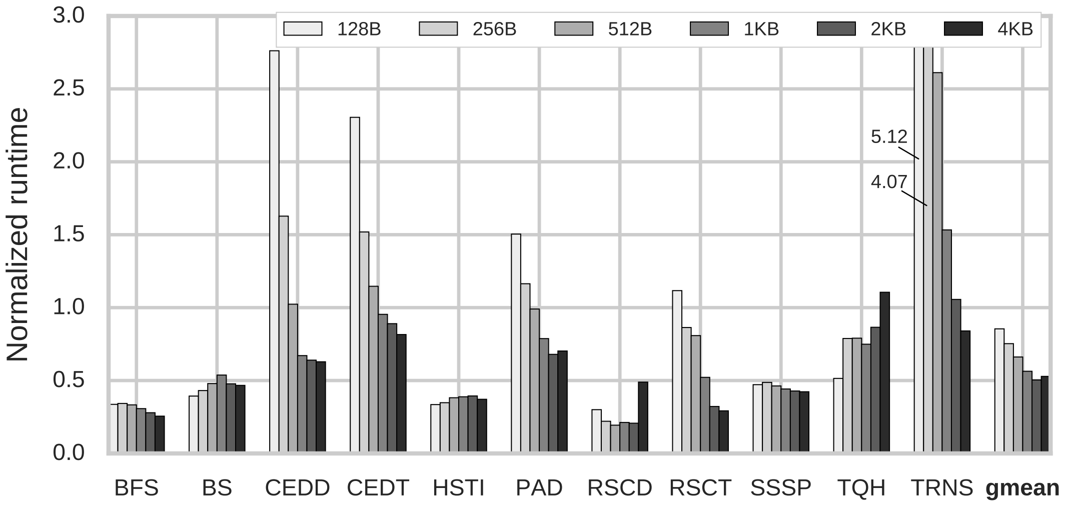


Figure 6.6: Execution time for various migration granularities normalized to the baseline demand paging scheme.

In *CEDD* the GPU uses only one input buffer that is recycled every frame. Recycling the same buffer improves execution time because only the first migration pays the full long-latency page fault. *CEDT* divides the computation in stages, processing the first two in the GPU and the second two in the CPU. In order to pipeline the algorithm one buffer per frame is used; each buffer is only copied once to the GPU, so our scheme cannot avoid many long-latency page faults. Still, the 4KB configuration achieves 17% lower execution time than the baseline due to the reduced latency when migrating data back to the CPU. In addition, we avoid several page faults on the pages containing the synchronization variables, which are migrated back and forth as GPU and CPU update them to coordinate the work.

TRNS performs an in-place matrix transposition and splits the work with a coarse-grained data partitioning scheme. *PAD* does an in-place padding operation on a matrix, partitioning the matrix in blocks that are dynamically assigned to GPU and CPU threads at runtime. Both benchmarks are memory bound and operate on large blocks of contiguous memory; consequently, fine-grained migrations struggle to match the performance of the more efficient page-sized migrations. In Section 6.5.3 we analyze how the latency of the interconnect affects these benchmarks and whether fine-grained migrations are feasible. In both benchmarks the 4KB configuration reduces the execution time over the baseline because our scheme decreases the latency of migrations for already known pages, as well as for migrations back to the CPU.

Figure 6.7 shows the number of total (non aggregated) migrations normalized to the number of migrations on the 128B configuration. In an ideal scenario where GPU and CPU access contiguous memory regions and there is no false sharing, doubling the migration

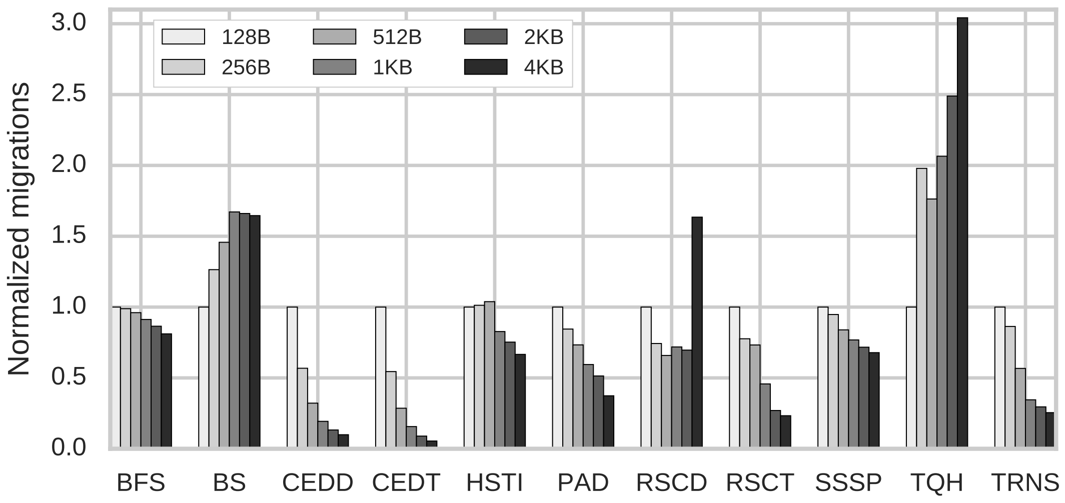


Figure 6.7: Number of total migrations with various migration granularities normalized to the configuration with 128B migrations.

size would halve the number of migrations required. If, on the other hand, the number of operations increases, it can be attributed to false sharing. False sharing occurs when GPU and CPU are simultaneously reading or writing two physical addresses that are located within a migration range; this causes a ping-pong effect where the data in that range is copied back and forth between the two memories. The larger the migration size is, the more likely it is for false sharing to occur. We see in Figure 6.7 how *BS*, *HSTI*, *RSCD* and *TQH* require additional DMA operations as we increase the size of migrations, which indicates the benchmarks suffer from false sharing.

RSCD and *RSCT* implement the same consensus algorithm with different partitioning of the work: data-partitioning in the case of *RSCD* and task-partitioning on *RSCT*. In *RSCD* both GPU and CPU iterate on a loop selecting two random flow vectors and estimating the parameters of a mathematical model; on every iteration 256 and 128 bytes of parameters are read respectively, as well as two random 16-byte flow vectors. The most efficient migration size for this benchmark is somewhere between 256 and 512 bytes, as larger migrations are likely to cause false sharing. Yet, Figure 6.6 shows that although the 512B configuration achieves the best results with an 80% execution time reduction over the baseline, all other configurations except 128B and 4KB follow closely.

Figure 6.7 shows how the number of migrations required decreases as we increase the migration size from 128B to 512B. On the other hand, larger transfer sizes increase the total number of migrations required, a clear sign of false sharing. The performance is not degraded although the benchmark suffers from false sharing because larger migrations prefetch model parameters for future iterations; the GPU consumes flow vectors much faster than the CPU and is able to use the prefetched data before the CPU migrates it away.

The benchmark obtains such a large speedup because only a few pages are migrated multiple times back and forth between the memories, and we significantly reduce the latency of most migrations. *RSCT* partitions the work by tasks, where the CPU threads calculate the model parameters and the GPU evaluates the model. In this implementation the GPU evaluates flow vectors in sequential order instead of randomly; large migrations are able to exploit spatial locality achieving a better performance than fine-grained data transfers.

BS displays a fine-grained memory access pattern where CPU and GPU threads iterate on a loop and are dynamically assigned points in a 3-dimensional space. Figure 6.7 shows how fine-grained migrations are more efficient, as larger granularities incur false sharing and require additional data transfers. In the end even the 4KB configuration achieves a significant 51% reduction in execution time over the baseline because our scheme avoids most of the long-latency page faults.

HSTI performs a histogram of the pixel values in a monochrome image. The bins are padded to lie in different cache lines; the 128B configuration is very efficient because on every atomic increment it migrates only the cache line where the bin is. Large migrations are expected to cause false sharing and a significant slowdown as GPU and CPU contend for the cache lines containing the bins, but that is not the case. The reason, as we saw in Section 5.3 is that the GPU with a large number of threads is much faster processing the image, and large migration sizes prefetch bins that other GPU threads will increment. In the end the 128B configuration performs best with a 65% execution time reduction over the baseline.

TQH also implements an image histogram using work queues in a producer-consumer model. Four CPU threads read and insert pixels in work queues; the GPU reads the pixels and performs the histogram. The benchmark uses several queue counter variables to synchronize the work; migration sizes beyond 128 bytes incur false sharing of the blocks containing these counters, increasing the number of migrations required. There is a large number of long-latency page faults we cannot avoid corresponding to the image pixels, as they are migrated only once to the GPU; this causes a 10% slowdown in our scheme with 4KB migrations due to the additional latency on handling first-touch page faults compared to the baseline. In addition, the pixels are never migrated back to the CPU and thus the benchmark does not benefit from the faster migrations to the CPU our scheme provides. The 128B configuration is able to achieve 49% lower execution time and is the most efficient by migrating the least amount of data.

BFS and *SSSP* are two graph traversal algorithms that switch computation between GPU and CPU at every frontier depending on its size; large frontiers are more efficiently computed on the GPU, while smaller are on the CPU. The number of nodes in each frontier is always high enough so that large migration sizes are more efficient. Figure 6.2 shows that both benchmarks migrate a lot of unnecessary data with 4KB migrations, but since the computation is not concurrent and switches between GPU and CPU, it does not cause migrations to actively steal data being used by the other and thus performance is not degraded. Indeed, Figure 6.7 shows no additional DMA operations are performed as we increase the granularity of migrations. Both benchmarks obtain a considerable speedup because there are numerous migrations of the same data between the two memories, and our scheme significantly reduces their latency.

6.5.2 Impact of Block Sizes in Data Migrations

As discussed in Section 3.2, choosing an optimal block size is a complex problem that has been the subject of many studies in the last decades. Small block sizes enable better distribution of the work among computing elements, but tend to be burdened by the overhead of thread creation on shared-memory multicore machines. Large block sizes, on the other hand, amortize the costs over longer computations, but can create load imbalance in the system and underutilize computing resources.

In heterogeneous architectures, the block size determines the granularity at which data migrates between host and device. Understandably, it has significant effect on the efficiency of migrations, the amount of data that is unnecessarily migrated and the amount of false sharing that may occur. As an example, in our initial experiments *PAD* was configured with a smaller block size, causing false sharing with migrations larger than 256 bytes that severely degraded performance.

The issue is exacerbated in heterogeneous architectures because they combine computing elements with different characteristics (instruction-level parallelism vs. thread-level parallelism) and hence different optimal block sizes. Strategies to efficiently partition work and data in heterogeneous architectures are out of the scope of this work, as a whole new dissertation could be written on the topic. Still, our goal of providing efficient data sharing between GPU and CPU at small granularities can allow programmers to more efficient partition the work in fine granularities.

Overall, *BFS*, *CEDD*, *CEDT*, *RSCT*, *SSSP*, and *TRNS* perform best with 4KB migrations because they do not show fine-grained data sharing between GPU and CPU, and thus large migrations are more efficient to amortize the latency of data transfers. *BS*, *HSTI*, *RSCD* and *TQH* show various degrees of sensitivity to migration sizes, and tend to perform best with fine grain data transfers that avoid false sharing.

6.5.3 Link Latency Analysis

Fine grain migrations are desirable to avoid false sharing and unnecessary data transfers, but the results from Section 6.5.1 indicate that the overheads are too high when cache line-sized migrations are used. The main source of overhead is the PCIe link, with a RTT of 2μs. A 4KB page migrated one line at at time can take up to 64μs with current link speeds, a latency not even a highly-threaded GPU can hide. Fortunately, the new generation of interconnects such as NVLink [126] reduce this latency and will perhaps make small data transfers practical. There is no public information available regarding the exact round-trip time of NVLink, but NVIDIA's whitepaper claims it is between 5 and 12 times faster than PCIe. In order to evaluate how changes in the interconnect latency will affect our scheme and whether cache line-sized migrations are feasible, we run all benchmarks with latencies going from the 2μs of PCIe 3.0 to a RTT of 0.1μs.

Figure 6.8 shows how execution time varies as a function of the link latency for various migration granularities. For every latency the results are normalized to the baseline demand paging scheme and that same latency. An interesting effect of varying the link latency can be seen in the benchmarks that suffer more from false sharing: *BS*, *RSCD* and *TQH*. Different latencies modify the timings and therefore the data interleaving between CPU and GPU, which can aggravate or alleviate the impact of false sharing. *RSCD* and *TQH* see their curve smoothing on the 0.1μs configuration; faster migrations increase the time data is available in one memory from request until it is migrated away, and thus false sharing is reduced.

BS still suffers from false sharing with large migrations. Yet, as we decrease link latency the benefits of prefetching data start to offset the overheads of false sharing, and larger granularities achieve better performance. *BFS* and *SSSP* obtain speedups with our scheme due to faster data transfers when switching computation between CPU and GPU. This effect is consistent as we reduce link latency and therefore the benchmarks show little variation.

CEDD, *CEDT*, *PAD*, *RSCT*, *TRNS* access large blocks of sequential data, and as we saw in Figure 6.6 fine-grained migrations significantly degrade performance. Figure 6.8 shows how reducing the link latency closes the gap considerably. With a 0.1μs interconnect *CEDD* and *PAD* are able to match or slightly outperform the baseline configuration with cache line-sized migrations, while *CEDT* suffers a 10% slowdown that is recovered with 256B migrations. Overall, all benchmarks but *TRNS* achieve speedups or just break even with 256 byte-sized migrations and a 0.5μs link latency. This indicates that fine-grained migrations will be possible as long as future interconnects provide latencies 4 to 5 times lower than current PCIe 3.0.

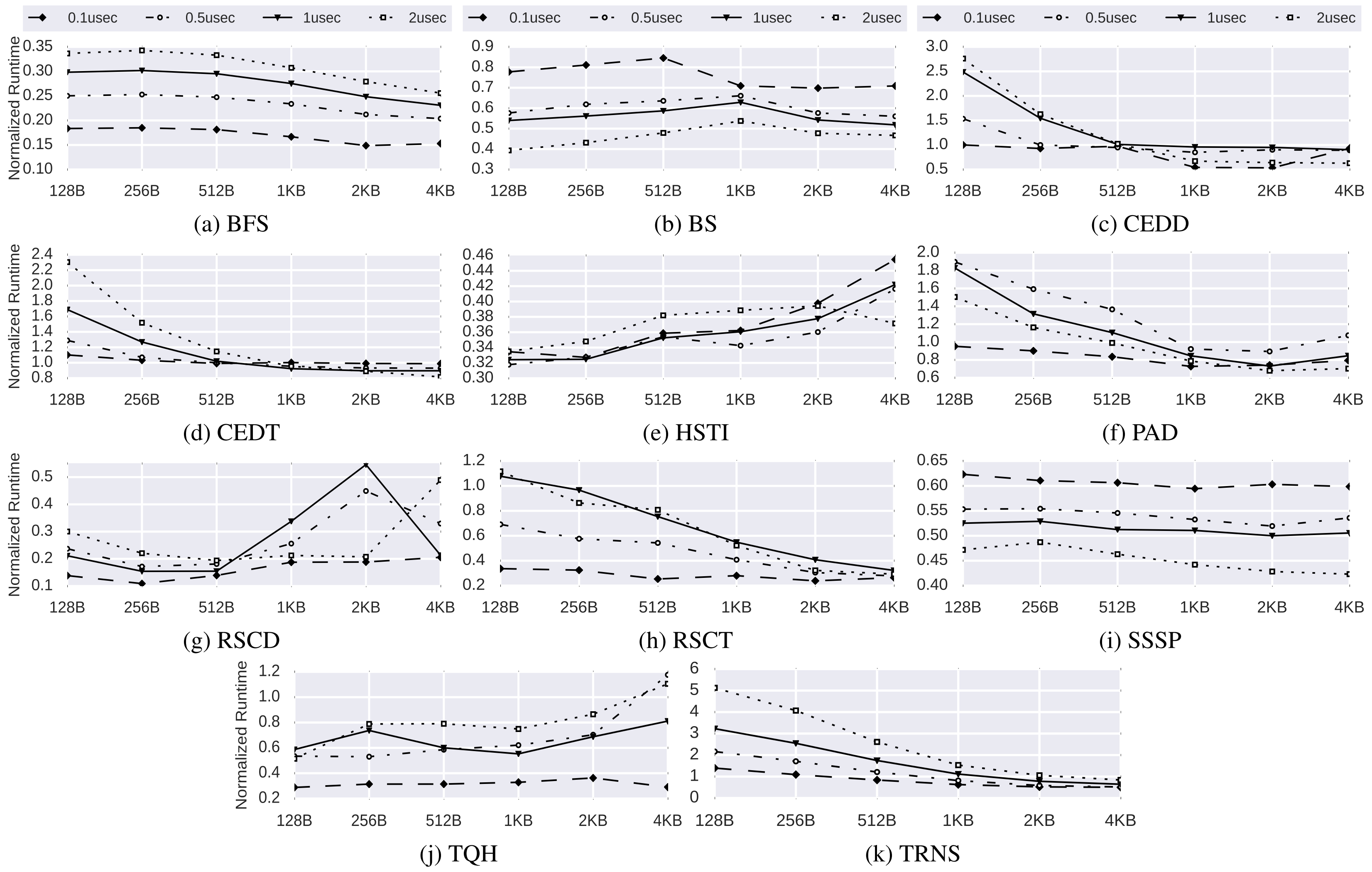


Figure 6.8: Execution time for various link latencies and migration granularities. Each configuration is normalized to the baseline demand paging scheme with that same latency.

6.6 Summary and Concluding Remarks

In this chapter we tackle the challenges and inefficiencies of demand paging in GPUs. We have shown how demand paging as currently implemented in Pascal-based GPUs is inefficient because GPUs are not able to resolve their own page faults without host intervention and must forward them to be handled by the CUDA runtime. This inefficiencies are exacerbated when executing collaborative computations where data migrates multiple times between host and device and the full page fault handling latency must be paid every time.

We have identified the issues of false sharing and unnecessary data transfers derived from the granularity at which data is migrated. In order to solve these problems we propose a memory organization and dynamic migration scheme that efficiently shares data between GPU and CPU memories at cache line-granularity. We leverage the observation that the page table of the heterogeneous process is rarely modified during runtime to reduce the migration latency of data within *known-pages*. In our scheme, only the first GPU access to a page in CPU memory incurs a page fault; following migrations can be done without software intervention and transparently to the operating system.

We evaluated our scheme with a set of collaborative benchmarks and found it reduces execution time by 15% on average with cache line size migrations, at the cost of degrading performance on benchmarks in which large blocks of contiguous memory are accessed. Although inefficient, we found large migration sizes achieve better performance due to the overheads of the PCIe interconnect. Our scheme with page-sized migrations obtains 47% lower execution times on average over the baseline demand paging system.

In order to understand whether smaller migrations are feasible on faster interconnect technologies, we evaluated all benchmarks with various link latencies and found that an interconnect with a round-trip time four to five times faster than PCIe is sufficient to efficiently perform fine-grained migrations. This leads us to conclude that fine-grained migrations will be feasible in future heterogeneous architectures connecting host and device via low-latency interconnects.

Chapter 7

Conclusion and Future Work

7.1 Conclusions

Efficient management of the memory subsystem in modern architectures is becoming an increasingly challenging task. Memory hierarchies have evolved to include several levels of on-chip caches and gigabytes of off-chip DRAM, that must now feed tens of data-hungry cores working at double or triple the speed than the memories themselves.

The task becomes yet more challenging with the emergence of heterogeneous architectures combining GPUs with traditional general-purpose cores, either integrated on the same die or connected through an expansion bus. These systems have become commonplace in the field of HPC due to the enormous computing potential that GPUs offer.

The trend towards tighter integration of GPU and CPU cores opens the design space for a new kind of heterogeneous computations, where both host and device collaborate on the computation, sharing data at fine granularities and synchronizing via system-wide atomic operations. Collaborative heterogeneous computations have only recently started to receive attention from the research community, but we believe they will pose a paradigm shift that will shape the heterogeneous architectures of the future.

The impact on the memory subsystem of combining cores with different characteristics requires a thorough analysis to understand the challenges and inefficiencies they suffer from, compared to homogeneous machines. This dissertation analyzes some of the challenges of efficiently managing the memory subsystem of modern systems, focusing in particular in the use of new and emerging programming models and computational paradigms, and proposes new techniques to guide the design of future architectures.

Our first contribution on this thesis is a prefetching scheme for SMPs systems that avoids most of the typical problems associated with prefetching. Our scheme relies on the runtime system of a task-based programming model to guide prefetching. By using a runtime system with knowledge of the data required by each task, we prefetch only useful data, avoiding speculation and thus cache pollution derived from mispredictions. That knowledge allows the runtime to prefetch blocks of data of variable size, reducing the prefetch instruction overhead compared to traditional software-based prefetch schemes that prefetch one line at a time. The runtime dynamically adapts and selects the best cache level to prefetch into, based on the estimated free space each cache level has at any point in time. In this manner our scheme avoids evicting data that is still useful, reducing cache thrashing.

We then turn our focus to heterogeneous architectures combining GPU and CPU cores. The second contribution is an in-depth analysis on the impact of sharing the last-level cache on a system integrating the GPU on die. We argue that the current literature on resource sharing on heterogeneous architectures does not evaluate collaborative computations, and therefore cannot provide enough insights on the challenges of resource sharing within these environments. We show how sharing the LLC is beneficial when applications share data between GPU and CPU during the computation, as a shared LLC allows for faster communication and synchronization. In addition, sharing the LLC guarantees a better utilization of the cache, as both GPU and CPU can access the full cache space if needed.

Our third contribution is a memory organization and data migration scheme for heterogeneous architectures with discrete GPUs. We show how demand paging as currently implemented in Pascal-based GPUs is inefficient. GPUs are not yet able to handle their own page faults and must forward them to the CUDA runtime running on the host, introducing overheads that result in significant slowdowns. We also show how migrating full memory pages on every access is inefficient and may cause false sharing, further degrading performance.

Then, we propose a memory organization that migrates data between CPU and GPU memories at fine granularities and without software intervention. By avoiding the need to pay the latency of GPU-initiated page faults on every migration, we improve performance on collaborative computations in which data migrates back and forth between the two memories. We show how fine-grained migrations suffer when the GPU is connected to the host via a high-latency interconnect such as PCIe. Finally, we analyze how our scheme would perform with future interconnects that reduce the round-trip time, and conclude that fine grain migrations are feasible as long as the interconnect is four to five times faster than PCIe 3.0.

7.2 Future Work

The work done in this dissertation leaves several lines of research to further explore efficient memory management, both on SMPs and on systems with GPUs. In the following section we detail some potential future work that could continue the research done throughout the thesis.

7.2.1 Runtime-Assisted Prefetching

Using the runtime system to guide prefetching allows for sophisticated techniques usually not possible with only hardware-based prefetchers. The amount of information available to the runtime system will dictate how efficient prefetching is. In our experiments we noticed that although the runtime system of OmpSs has knowledge about input and output data used by each tasks, as declared by the user, that may not be enough. Tasks are allowed to allocate their own local data in the stack, and therefore the information to decide dynamically where to prefetch into may be incomplete. Since stack-based memory allocation is static, a potential improvement would be to adapt the compiler to provide the runtime system with information about the static memory allocated by each task.

Another line of research that can be explored is using idle threads to prefetch for other cores. In our current implementation a core only prefetches data for a task that it will execute in the future. Due to data dependencies, there may not be enough parallelism to keep all cores busy at all time. An idle core may start prefetching data for a task that due to the affinity scheduling policy will be executed by a different core. Most systems include a shared cache level where data could be fetched into from off-chip memory and used by a different core than that starting the prefetch. This could be specially interesting on heterogeneous systems with cores of different sizes, such as in ARM's big.LITTLE designs. In this manner, a small core could start prefetching data for the big core that will execute the task in the future.

7.2.2 Resource Sharing on Integrated Systems

Resource sharing has only recently started to attract some attention from researchers. The work we have done in this thesis evaluating resource sharing with collaborative computations is, to the best of our knowledge, the first to evaluate integrated heterogeneous architectures with such computations. Further research is required to understand how other shared resources, *e.g.* the memory controllers or the NoC are affected when both GPU and CPU cores work together sharing data and synchronizing.

Our experiments assume a strict consistency model similar to that found on CPUs. It is not clear what effect this has on the cache hierarchy and in particular on the shared cache level. More relaxed models such as those found on GPUs could be evaluated, analyzing the impact they have on the cache hierarchy when executing collaborative computations with a lot of data sharing and synchronization. In addition, the architecture we evaluate in this work uses the MESI coherence protocol throughout the system. Further research could be done exploring the impact of using more advanced protocols or even hybrid protocols where GPU and CPU caches are kept coherent with different states.

7.2.3 Efficient Data Sharing on Heterogeneous Architectures

The data migration scheme we propose attempts to avoid the inefficiencies of the demand paging implementation currently found in NVIDIA GPUs. Our work assumes that GPUs cannot execute their own page fault handling routines, and must therefore forward them to the CPU. Recent work in the literature proposes mechanisms to allow GPUs to context switch and potentially resolve their own page faults. This is an interesting line of research to explore, as most of the overheads of the current demand paging scheme could be reduced by avoiding host intervention.

A different way to tackle the problem, more in line with the approach we propose in this work would be to move away from paging altogether. Reducing the granularity of migrations to cache lines as we have seen provides benefits as long as the interconnect supports it. A potential line of research would be to treat GPU memory as a cache, completely removing the local page table in a similar manner to integrated architectures. Although there are works in the literature proposing a similar approach, none consider collaborative computations. The memory access pattern of these computations greatly differ to that of traditional heterogeneous kernels, where data is copied in bulk transfers before and after the computation. Further research is required to understand how a memory system where the GPU's memory is treated as a DRAM cache would behave when executing collaborative computations.

Data prefetching is another interesting line of research that could continue the memory organization and data migration scheme work we propose in this thesis. We have seen how fine-grained data migration is only efficient when a low-latency interconnect is used, but prefetching data in advance could hide some of the latency and make fine-grained migrations feasible even on current interconnect technologies. The CUDA API already provides some prefetching hints that can be used by programmers to help the driver with automatic data movement. Combining the existing prefetching hints with a fine-grained migration scheme could be an interesting line of research to continue our work.

Appendix A

Publications

A.1 Thesis Related Publications

- *"Efficient data sharing on heterogeneous systems"*. Víctor García Flores, Eduard Ayguade, and Antonio J. Peña. In 46th International Conference on Parallel Processing (ICPP). Bristol, United Kingdom. August 2017.

- *"Adaptive runtime-assisted block prefetching on chip-multiprocessors"*. Víctor García Flores, Alejandro Rico, Carlos Villavieja, Paul Carpenter, Nacho Navarro, and Alex Ramirez. International Journal of Parallel Programming, Vol. 45(3). June 2017.

- *"Evaluating the effect of last-level cache sharing on integrated GPU-CPU systems with heterogeneous applications"*. Víctor García Flores, Juan Gómez Luna, Thomas Grass, Alejandro Rico, Eduard Ayguade, and Antonio J. Peña. In International Symposium on Workload Characterization (IISWC). Providence, Rhode Island. September 2016.

- *"Analyzing the effect of last level cache sharing on integrated platforms with fine-grain CPU-GPU collaboration"*. Víctor García Flores, Juan Gómez Luna, Thomas Grass, Eduard Ayguade, and Antonio J. Peña. In GPU Technology Conference Europe (GTC Europe). Amsterdam, September 2016. Poster.

- *"Adaptive runtime-assisted block prefetching on chip-multiprocessors"*. Víctor García Flores, Alejandro Rico, Carlos Villavieja, Paul Carpenter, Nacho Navarro, and Alex Ramirez. In On-chip Memory Hierarchies and Interconnects Workshop. Porto, Portugal, August 2014.

A.2 Other Publications

- *"Chai: Collaborative heterogeneous applications for integrated-architectures"*. Juan Gómez-Luna, Izzat El Hajj, Li-Wen Chang, Víctor García Flores, Simon Garcia de Gonzalo, Thomas Jablin, Antonio J. Peña, and Wen-mei Hwu. In Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). San Francisco, CA, USA, April 2017.

- *"The data transfer engine: Towards a software controlled memory hierarchy"*. Víctor García Flores, Alejandro Rico, Carlos Villavieja, Nacho Navarro, and Alex Ramirez. In Advanced Computer Architecture and Compilation for Embedded Systems (ACACES). Fiuggi, Italy, July 2012. Poster Abstract, pp. 215–218.

- *"Architecture for a million core processor"*. Zeus Gomez Marmolejo, Víctor García Flores, Alex Ramirez, and Nacho Navarro. In Advanced Computer Architecture and Compilation for Embedded Systems (ACACES). Fiuggi, Italy, July 2011. Poster Abstract, pp. 245–248.

- *"Bringing the multi-core paradigm to OS design"*. Víctor García Flores, Zeus Gomez Marmolejo, Alex Ramirez, and Nacho Navarro. In Advanced Computer Architecture and Compilation for Embedded Systems (ACACES). Terrassa, Spain, July 2010. Poster Abstract, p. 255–258.

Bibliography

[1] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach. 5th edn.* Elsevier, 2012.

[2] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995.

[3] T. Mudge, "Power: A first-class architectural design constraint," *Computer*, vol. 34, no. 4, pp. 52–58, Apr. 2001.

[4] Top500. (2016) The Top500 list of supercomputers. [Online]. Available: https://www.top500.org/lists/2016/11/

[5] I. Corporation. (2015) Optimizing application performance on intel core microarchitecture using hardware-implemented prefetchers. [Online]. Available: https://software.intel.com/en-us/articles/optimizing-application-performance-on-intel-coret-microarchitecture-using-hardware\-implemented-prefetchers

[6] S. Byna, Y. Chen, and X.-H. Sun, "A taxonomy of data prefetching mechanisms," ser. ISPAN '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 19–24.

[7] J. Lee, H. Kim, and R. Vuduc, "When prefetching works, when it doesnt, and why," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 1, pp. 2:1–2:29, Mar. 2012.

[8] T. R. Puzak, A. Hartstein, P. G. Emma, and V. Srinivasan, "When prefetching improves/degrades performance," in *Proceedings of the 2nd conference on Computing frontiers*, ser. CF '05. New York, NY, USA: ACM, 2005, pp. 342–352.

[9] O. Consortium. Openmp website. [Online]. Available: http://openmp.org/wp/

[10] 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. 2, pp. 173–193, 2011.

[11] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, ser. PLDI '98. New York, NY, USA: ACM, 1998, pp. 212–223.

[12] L. Bergstrom, "Measuring NUMA effects with the STREAM benchmark," *CoRR*, vol. abs/1103.3225, 2011. [Online]. Available: http://arxiv.org/abs/1103.3225

[13] B. Jeff, "Big.little system architecture from arm: saving power through heterogeneous multiprocessing and task context migration." in *DAC*, P. Groeneveld, D. Sciuto, and S. Hassoun, Eds. ACM, 2012, pp. 1143–1146. [Online]. Available: http://dblp.uni-trier.de/db/conf/dac/dac2012.html#Jeff12

[14] *Compute Cores. Whitepaper*, AMD, 2014. [Online]. Available: https://www.amd.com/Documents/Compute_Cores_Whitepaper.pdf

[15] Green500. (2016) The Green500 list of the most energy-efficient supercomputers. [Online]. Available: https://www.top500.org/green500/list/2016/06/

[16] Intel Corporation. (2015) The compute architecture of Intel processor graphics Gen9. [Online]. Available: https://software.intel.com/sites/default/files/managed/c5/9a/The-Compute-Architecture-of-Intel-Processor-Graphics-Gen9-v1d0.pdf

[17] NVIDIA. (2011) New CUDA 4.0 release makes parallel programming easier. [Online]. Available: http://www.nvidia.co.uk/object/nvidia-cuda-4-0-press-20110228-uk.html

[18] ——. (2016) CUDA 8 features revealed. [Online]. Available: https://devblogs.nvidia.com/parallelforall/cuda-8-features-revealed/

[19] AMD. (2012) Heterogeneous System Architecture: A Technical Review. [Online]. Available: http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/hsa10.pdf

[20] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: programming the memory hierarchy," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006.

[21] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 2, pp. 187–198, Feb. 2011.

[22] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 519–538.

[23] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, Aug. 2007.

[24] J. Reinders, *Intel threading building blocks*, 1st ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.

[25] A. J. Smith, “Cache memories,” *ACM Comput. Surv.*, vol. 14, no. 3, pp. 473–530, Sep. 1982.

[26] R. Lee, P.-C. Yew, and D. Lawrie, *Data prefetching in shared memory multiprocessors*, Jan 1987.

[27] J.-L. Baer and T.-F. Chen, “Effective hardware-based data prefetching for high-performance processors,” *IEEE Trans. Comput.*, vol. 44, no. 5, pp. 609–623, May 1995.

[28] D. Joseph and D. Grunwald, “Prefetching using markov predictors,” *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 121–133, Feb 1999.

[29] K. Nesbit and J. Smith, “Data cache prefetching using a global history buffer,” in *Software, IEE Proceedings-*, feb. 2004, p. 96.

[30] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, “Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers,” ser. HPCA ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 63–74.

[31] V. Jiménez, R. Gioiosa, F. J. Cazorla, A. Buyuktosunoglu, P. Bose, and F. P. O’Connell, “Making data prefetch smarter: Adaptive prefetching on power7,” in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’12. New York, NY, USA: ACM, 2012, pp. 137–146.

[32] G. developers. Gcc documentation. [Online]. Available: http://gcc.gnu.org/onlinedocs/gcc-4.0.4/gcc/Optimize-Options.html

[33] I. Corporation. x86 Instruction Set Reference. [Online]. Available: http://x86.renejeschke.de/html/file_module_x86_id_252.html

[34] A. Ltd. ARM Cortex-A Series Programmers Guide for ARMv8-A. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0024a/ch06s03s07.html

[35] E. H. Gornish, E. D. Granston, and A. V. Veidenbaum, “Compiler-directed data prefetching in multiprocessors with memory hierarchies,” in *In International Conference on Supercomputing*, 1990, pp. 354–368.

[36] T. Mowry and A. Gupta, “Tolerating latency through software-controlled prefetching in shared-memory multiprocessors,” *Journal of Parallel and Distributed Computing*, vol. 12, pp. 87–106, 1991.

[37] T. C. Mowry, M. S. Lam, and A. Gupta, “Design and evaluation of a compiler algorithm for prefetching,” in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS V. New York, NY, USA: ACM, 1992, pp. 62–73.

[38] T.-F. Chen and J.-L. Baer, “A performance study of software and hardware data prefetching schemes,” ser. ISCA ’94. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 223–232.

[39] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems, “Guided region prefetching: A cooperative hardware/software approach,” in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ser. ISCA ’03. New York, NY, USA: ACM, 2003, pp. 388–398.

[40] N. P. Jouppi, “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers,” *SIGARCH Comput. Archit. News*, vol. 18, no. 2SI, pp. 364–373, May 1990.

[41] M. Wall. (2001) Using block prefetch for optimized memory performance. [Online]. Available: http://web.mit.edu/ehliu/Public/ProjectX/Meetings/AMD_block_prefetch_paper.pdf

[42] ARM. (2008) Cortex-a9 technical reference manual. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0388i/CHDFCCIH.html

[43] V. Papaefstathiou, M. G. Katevenis, D. S. Nikolopoulos, and D. Pnevmatikatos, “Prefetching and cache management using task lifetimes,” in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ser. ICS ’13. New York, NY, USA: ACM, 2013, pp. 325–334.

[44] Qualcomm. (2013) Snapdragon S4 processors: System on chip solutions for a new mobile age. Whitepaper. [Online]. Available: https://www.qualcomm.com/documents/snapdragon-s4-processors-system-chip-solutions-new-mobile-age

[45] *Exynos 5. Whitepaper*, Samsung, 2012. [Online]. Available: http://www.samsung.com/global/business/semiconductor/minisite/Exynos/data/Enjoy_the_Ultimate_WQXGA_Solution_with_Exynos_5_Dual_WP.pdf

[46] NVIDIA. (2015) NVIDIA Tegra X1. [Online]. Available: http://www.nvidia.com/object/tegra-x1-processor.html

[47] Intel Corporation. (2013) Products (formerly Haswell). [Online]. Available: http://ark.intel.com/products/codename/42174/Haswell

[48] *GNC Architecture. Whitepaper*, AMD, 2012. [Online]. Available: https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf

[49] M. Daga, A. M. Aji, and W. Feng, “On the efficacy of a fused CPU+GPU processor (or APU) for parallel computing,” in *Symposium on Application Accelerators in High-Performance Computing*, 2011.

[50] M. Daga and M. Nutter, “Exploiting coarse-grained parallelism in B+ tree searches on an APU,” in *SC Companion: High Performance Computing, Networking Storage and Analysis (SCC)*, 2012.

[51] M. Daga, M. Nutter, and M. Meswani, “Efficient breadth-first search on a heterogeneous processor,” in *IEEE International Conference on Big Data*, Oct. 2014, pp. 373–382.

[52] M. C. Delorme, T. S. Abdelrahman, and C. Zhao, “Parallel radix sort on the amd fusion accelerated processing unit,” in *42nd International Conference on Parallel Processing*, Oct. 2013, pp. 339–348.

[53] J. He, M. Lu, and B. He, “Revisiting co-processing for hash joins on the coupled CPU-GPU architecture,” *Proc. VLDB Endow.*, vol. 6, no. 10, pp. 889–900, Aug. 2013.

[54] J. Hestness, S. W. Keckler, and D. A. Wood, “GPU computing pipeline inefficiencies and optimization opportunities in heterogeneous CPU-GPU processors,” in *IEEE International Symposium on Workload Characterization (IISWC)*, 2015, pp. 87–97.

[55] W. Liu and B. Vinter, “Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors,” *Parallel Comput.*, vol. 49, no. C, pp. 179–193, Nov. 2015.

[56] J. Lee and H. Kim, “Tap: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture,” in *IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA)*, 2012.

[57] V. Mekkat, A. Holey, P.-C. Yew, and A. Zhai, “Managing shared last-level cache in a heterogeneous multicore processor,” in *International Conference on Parallel Architectures and Compilation Techniques*, 2013.

[58] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, “Adaptive insertion policies for high performance caching,” in *34th International Symposium on Computer Architecture (ISCA)*, 2007, pp. 381–391.

[59] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu, “Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems,” in *39th International Symposium on Computer Architecture (ISCA)*, 2012, pp. 416–427.

[60] O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, “Managing GPU concurrency in heterogeneous architectures,” in *47th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014, pp. 114–126.

[61] Khronos group, “OpenCL,” http://www.khronos.org/opencl/, 2011.

[62] *The OpenCL Specification v2.0*, Khronos OpenCL Working Group, 2015. [Online]. Available: https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf

[63] NVIDIA. (2013) Unified memory in CUDA 6. [Online]. Available: https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/

[64] ——. (2016) The future of Unified Memory. [Online]. Available: http://on-demand.gputechconf.com/gtc/2016/presentation/s6216-nikolay-sakharnykh-future-unified-memory.pdf

[65] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, "An investigation of unified memory access performance in CUDA," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2014.

[66] W. Li, G. Jin, X. Cui, and S. See, "An evaluation of unified memory technology on NVIDIA GPUs," in *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2015.

[67] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch, "Unlocking bandwidth for GPUs in CC-NUMA systems," in *Int. Symp. on High Performance Computer Architecture (HPCA)*, 2015.

[68] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "Page placement strategies for GPUs within heterogeneous memory systems," in *ACM SIGPLAN Notices*, vol. 50, no. 4, 2015, pp. 607–618.

[69] J. Kehne, J. Metter, and F. Bellosa, "GPUswap: Enabling oversubscription of GPU memory through transparent swapping," in *ACM SIGPLAN/SIGOPS Int. Conf. on Virtual Execution Environments*, 2015.

[70] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler, "Towards high performance paged memory for GPUs," in *IEEE International Symp. on High Performance Computer Architecture*, 2016.

[71] S. Shahar, S. Bergman, and M. Silberstein, "ActivePointers: A case for software address translation on GPUs," in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, 2016.

[72] Y. Kim, J. Lee, J. E. Jo, and J. Kim, "GPUdmm: A high-performance and memory-oblivious GPU architecture using dynamic memory management," in *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[73] AMD. (2015) High Bandwidth Memory (HBM). [Online]. Available: https://www.amd.com/Documents/High-Bandwidth-Memory-HBM.pdf

[74] NVIDIA. (2016) NVIDIA Tesla P100. [Online]. Available: http://images.nvidia.com/content/tesla/pdf/nvidia-teslap100-techoverview.pdf

[75] B. Wang, B. Wu, D. Li, X. Shen, W. Yu, Y. Jiao, and J. S. Vetter, "Exploring hybrid memory for GPU energy efficiency through software-hardware co-design," in *Proc. of the 22nd International Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[76] J. Zhao, G. Sun, G. H. Loh, and Y. Xie, “Optimizing GPU energy efficiency with 3D die-stacking graphics memory and reconfigurable memory interface,” *ACM Trans. Archit. Code Optim.*, 2013.

[77] L. Zhao, R. Iyer, R. Illikkal, and D. Newell, “Exploring DRAM cache architectures for CMP server platforms,” in *25th International Conference on Computer Design*, 2007.

[78] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian, “CHOP: Adaptive filter-based DRAM caching for CMP server platforms,” in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2010.

[79] M. K. Qureshi and G. H. Loh, “Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical SRAM-tags with a simple and practical design,” in *45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.

[80] J. Sim, G. H. Loh, H. Kim, M. O’Connor, and M. Thottethodi, “A mostly-clean DRAM cache for effective hit speculation and self-balancing dispatch,” in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.

[81] D. Jevdjic, S. Volos, and B. Falsafi, “Die-stacked DRAM caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[82] C. Chou, A. Jaleel, and M. K. Qureshi, “CAMEO: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache,” in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[83] L. Lamport, “How to make a multiprocessor computer that correctly executes multiprocess programs,” *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 690–691, Sep. 1979.

[84] S. Owens, S. Sarkar, and P. Sewell, “A better x86 memory model: X86-TSO,” in *22Nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 2009, pp. 391–407.

[85] S. V. Adve and K. Gharachorloo, “Shared memory consistency models: A tutorial,” *Computer*, vol. 29, no. 12, pp. 66–76, Dec. 1996.

[86] J. Goodacre and A. N. Sloss, “Parallelism and the ARM instruction set architecture,” *Computer*, vol. 38, no. 7, pp. 42–50, Jul. 2005.

[87] B. A. Hechtman and D. J. Sorin, “Exploring memory consistency for massively-threaded throughput-oriented processors,” in *40th International Symposium on Computer Architecture (ISCA)*, 2013, pp. 201–212.

[88] *CUDA C Programming Guide*, NVIDIA Corporation, 2014. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[89] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous system coherence for integrated CPU-GPU systems," in *46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 457–467.

[90] A. Rico, F. Cabarcas, C. Villavieja, M. Pavlovic, A. Vega, Y. Etsion, A. Ramirez, and M. Valero, "On the simulation of large-scale architectures using multiple application abstraction levels," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 36:1–36:20, Jan. 2012.

[91] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200.

[92] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A heterogeneous CPU-GPU simulator," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 34–36, Jan. 2015.

[93] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[94] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *International Symposium on Performance Analysis of Systems and Software*, 2009.

[95] N. Muralimanohar and R. Balasubramonian, "Cacti 6.0: A tool to understand large caches," 2007.

[96] X. Feng, K. W. Cameron, and D. A. Buell, "Pbpi: a high performance implementation of bayesian phylogenetic inference," ser. SC '06. New York, NY, USA: ACM, 2006.

[97] I.-H. Chung and J. Hollingsworth, "A case study using automatic performance tuning for large-scale scientific programs," ser. HPDC '06, 2006, pp. 45–56.

[98] D. Lowenthal and M. James, "Run-time selection of block size in pipelined parallel programs," in *Parallel Processing, 1999. 13th International and 10th Symposium on Parallel and Distributed Processing, 1999. 1999 IPPS/SPDP. Proceedings*, Apr, pp. 82–87.

[99] A. Rico, A. Ramirez, and M. Valero, "Available task-level parallelism on the cell be," *Sci. Program.*, vol. 17, no. 1-2, pp. 59–76, Jan. 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1507443.1507448

[100] E. Rothberg, J. P. Singh, and A. Gupta, "Working sets, cache sizes, and node granularity issues for large-scale multiprocessors," ser. ISCA '93. New York, NY, USA: ACM, 1993, pp. 14–26.

[101] S. Tandri and T. Abdelrahman, "Automatic partitioning of data and computations on scalable shared memory multiprocessors," ser. ICPP '97, 1997, pp. 64–73.

[102] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Int. Symp. on Workload Characterization (IISWC)*, 2009.

[103] W.-M. Hwu, *Heterogeneous System Architecture: A new compute platform infrastructure*. Morgan Kaufmann, 2015.

[104] University of Rome "La Sapienza", "9th DIMACS Implementation Challenge," 2014, http://www.dis.uniroma1.it/challenge9/index.shtml.

[105] J. Gómez Luna, L.-W. Chang, I.-J. Sung, W.-M. Hwu, and N. Guil, "In-place data sliding algorithms for many-core architectures," in *44th International Conference on Parallel Processing (ICPP)*, Sep. 2015.

[106] AMD, "AMD accelerated parallel processing (APP) software development kit (SDK) 3.0," http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/, 2016.

[107] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides, and N. Guil, "An optimized approach to histogram computation on GPU," *Machine Vision and Applications*, vol. 24, no. 5, pp. 899–908, 2013.

[108] J. Gómez-Luna, H. Endt, W. Stechele, J. M. González-Linares, J. I. Benavides, and N. Guil, "Egomotion compensation and moving objects detection algorithm on GPU," in *Applications, Tools and Techniques on the Road to Exascale Computing*, ser. Advances in Parallel Computing, vol. 22. IOS Press, 2011, pp. 183–190.

[109] J. Gómez-Luna, I. E. Hajj, L.-W. Chang, V. Garcia-Flores, S. G. de Gonzalo, T. B. Jablin, A. J. Peña, and W.-M. Hwu, "Chai: Collaborative heterogeneous applications for integrated-architectures," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2017, pp. 1–10. [Online]. Available: https://chai-benchmarks.github.io/

[110] L. Piegl and W. Tiller, *The NURBS Book (2nd Ed.)*. New York, NY, USA: Springer-Verlag New York, Inc., 1997.

[111] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal, "Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory," ser. PACT '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 340–349.

[112] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems," *SIGARCH Comput. Archit. News*, vol. 38, no. 1, pp. 335–346, Mar. 2010.

[113] Y. Guo, P. Narayanan, M. Bennaser, S. Chheda, and C. Moritz, "Energy-efficient hardware data prefetching," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 19, no. 2, pp. 250–263, 2011.

[114] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver, "A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC," in *Design Automation Conference (DAC)*, 2012, pp. 850–855.

[115] J. Lee, S. Li, H. Kim, and S. Yalamanchili, "Adaptive virtual channel partitioning for network-on-chip in heterogeneous architectures," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 18, no. 4, pp. 48:1–48:28, 2013.

[116] N. Agarwal, T. Krishna, L. S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *International Symposium on Performance Analysis of Systems and Software*, 2009.

[117] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads," in *IEEE International Symposium on Workload Characterization (IISWC)*, Dec. 2010.

[118] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W. W. Hwu, "An asymmetric distributed shared memory model for heterogeneous parallel systems," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1. ACM, 2010, pp. 347–358.

[119] NVIDIA. (2016) GP100 Pascal Whitepaper. [Online]. Available: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[120] D. J. Miller, P. M. Watts, and A. W. Moore, "Motivating future interconnects: A differential measurement analysis of PCI latency," in *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2009.

[121] W. J. Bolosky and M. L. Scott, "False sharing and its effect on shared memory performance," in *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*, 1993.

[122] J. Torrellas, H. S. Lam, and J. L. Hennessy, "False sharing and spatial locality in multiprocessor caches," *IEEE Trans. on Computers*, 1994.

[123] Altera. (2009) Scatter-gather DMA controller core. [Online]. Available: https://www.altera.co.jp/content/dam/altera-www/global/ja_JP/pdfs/literature/hb/nios2/qts_qii55003.pdf

[124] ARM. (2005) PrimeCell DMA controller. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0196g/DDI0196.pdf

[125] M. Kistler, M. Perrone, and F. Petrini, "Cell multiprocessor communication network: Built for speed," *IEEE Micro*, vol. 26, no. 3, 2006.

[126] NVIDIA. (2014) Coral white paper. [Online]. Available: http://info.nvidianews.com/rs/nvidia/images/Coral%20White%20Paper%20Final-3-2.pdf