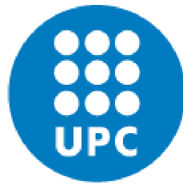


UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Multicore architecture optimizations for HPC applications

by
Uglješa Milić

ADVERTIMENT La consulta d'aquesta tesi queda condicionada a l'acceptació de les següents condicions d'ús: La difusió d'aquesta tesi per mitjà del repositori institucional UPCommons (http://upcommons.upc.edu/tesis) i el repositori cooperatiu TDX (http://www.tdx.cat/) ha estat autoritzada pels titulars dels drets de propietat intel·lectual **únicament per a usos privats** emmarcats en activitats d'investigació i docència. No s'autoritza la seva reproducció amb finalitats de lucre ni la seva difusió i posada a disposició des d'un lloc aliè al servei UPCommons o TDX. No s'autoritza la presentació del seu contingut en una finestra o marc aliè a UPCommons (*framing*). Aquesta reserva de drets afecta tant al resum de presentació de la tesi com als seus continguts. En la utilització o cita de parts de la tesi és obligat indicar el nom de la persona autora.

ADVERTENCIA La consulta de esta tesis queda condicionada a la aceptación de las siguientes condiciones de uso: La difusión de esta tesis por medio del repositorio institucional UPCommons (http://upcommons.upc.edu/tesis) y el repositorio cooperativo TDR (http://www.tdx.cat/?locale-attribute=es) ha sido autorizada por los titulares de los derechos de propiedad intelectual **únicamente para usos privados enmarcados** en actividades de investigación y docencia. No se autoriza su reproducción con finalidades de lucro ni su difusión y puesta a disposición desde un sitio ajeno al servicio UPCommons No se autoriza la presentación de su contenido en una ventana o marco ajeno a UPCommons (*framing*). Esta reserva de derechos afecta tanto al resumen de presentación de la tesis como a sus contenidos. En la utilización o cita de partes de la tesis es obligado indicar el nombre de la persona autora.

WARNING On having consulted this thesis you're accepting the following use conditions: Spreading this thesis by the institutional repository UPCommons (http://upcommons.upc.edu/tesis) and the cooperative repository TDX (http://www.tdx.cat/?locale-attribute=en) has been authorized by the titular of the intellectual property rights **only for private uses** placed in investigation and teaching activities. Reproduction with lucrative aims is not authorized neither its spreading nor availability from a site foreign to the UPCommons service. Introducing its content in a window or frame foreign to the UPCommons service is not authorized (framing). These rights affect to the presentation summary of the thesis as well as to its contents. In the using or citation of parts of the thesis it's obliged to indicate the name of the author.

Multicore Architecture Optimizations for HPC Applications

by

UGLJEŠA MILIĆ

A thesis submitted in fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Architecture

Departament d'Arquitectura de Computadors (DAC)
Universitat Politècnica de Catalunya (UPC)
Barcelona, Spain

Advisor: Dr. Alex Ramírez   *Co-advisor:* Dr. Paul M. Carpenter

September 2017

Abstract

From single-core CPUs to detachable compute accelerators, supercomputers made a tremendous progress by using available transistors on chip and specializing hardware for a given type of computation. Today, compute nodes used in HPC employ multi-core CPUs tailored for serial execution and multiple accelerators (many-core devices or GPUs) for throughput computing. However, designing next-generation HPC system requires not only the performance improvement but also better energy efficiency. Current trend of reaching exascale level of computation asks for at least an order of magnitude increase in both of these metrics.

This thesis explores HPC-specific optimizations in order to make better utilization of the available transistors and to improve performance by transparently executing parallel code across multiple GPU accelerators. First, we analyze several HPC benchmark suites, compare them against typical desktop applications, and identify the differences which advocate for proper core tailoring. Moreover, within the HPC applications, we evaluate serial and parallel code sections separately, resulting in an Asymmetric Chip Multiprocessor (ACMP) design with one core optimized for single-thread performance and many lean cores for parallel execution. Our results presented here suggests downsizing of core front-end structures providing an HPC-tailored lean core which saves 16% of the core area and 7% of power, without performance loss.

Further improving an ACMP design, we identify that multiple lean cores run the same code during parallel regions. This motivated us to evaluate the idea where lean cores share the I-cache with the intent of benefiting from mutual prefetching, without increasing the average access latency. Our exploration of the multiple parameters finds the sweet spot on a wide interconnect to access the shared I-cache and the inclusion of a few line buffers to provide the required bandwidth and latency to sustain performance. The projections presented in this thesis show

additional 11% area savings with a 5% energy reduction at no performance cost. These area and power savings might be attractive for many-core accelerators either for increasing the performance per area and power unit, or adding additional cores and thus improving the performance for the same hardware budget.

Finally, in this thesis we study the effects of future NUMA accelerators comprised of multiple GPU devices. Reaching the limits of a single-GPU die size, next-generation GPU compute accelerators will likely embrace multi-socket designs increasing the core count and memory bandwidth. However, maintaining the UMA behavior of a single-GPU in multi-GPU systems without code rewriting stands as a challenge. We investigate multi-socket NUMA GPU designs and show that significant changes are needed to both the GPU interconnect and cache architectures to achieve performance scalability. We show that application phase effects can be exploited allowing GPU sockets to dynamically optimize their individual interconnect and cache policies, minimizing the impact of NUMA effects. Our NUMA-aware GPU outperforms a single GPU by 1.5×, 2.3×, and 3.2× while achieving 89%, 84%, and 76% of theoretical application scalability in 2, 4, and 8 sockets designs respectively. Implementable today, NUMA-aware multi-socket GPUs may be a promising candidate for performance scaling of future compute nodes used in HPC.

Contents

Abstract iii

Contents iv

List of Figures ix

List of Tables xiii

1 Introduction 1
- 1.1 High Performance Computing . . . 1
- 1.2 Evolution of a Single Compute Node . . . 3
  - 1.2.1 Multicore Processors . . . 4
  - 1.2.2 Compute Accelerators . . . 5
  - 1.2.3 Multiple Compute Accelerators on a Single Node . . . 7
- 1.3 Programming Models for Single Node Architectures . . . 8
  - 1.3.1 OpenMP for Shared Memory Multicore Processors . . . 9
  - 1.3.2 OpenACC for Programming Accelerators . . . 10
  - 1.3.3 CUDA for Programming GPUs . . . 11
- 1.4 Thesis Contributions . . . 12
- 1.5 Timeline . . . 14
- 1.6 Thesis Organization . . . 15

2 Background and Related Work 17
- 2.1 From General-purpose to Specialized Systems-on-Chip . . . 17
- 2.2 Efficient CMP Design for HPC . . . 20
- 2.3 More Performance per Compute Node . . . 23

3 Methodology 27
- 3.1 Code Instrumentation . . . 27
- 3.2 Simulation Frameworks . . . 28
  - 3.2.1 Sniper . . . 28
  - 3.2.2 TaskSim . . . 29
  - 3.2.3 GPUSim . . . 31

3.3 Benchmark Suites . . . 32
3.3.1 Workloads for Shared-memory CMPs . . . 32
3.3.2 CUDA workloads for GPU analysis . . . 35

4 HPC Workload Characterization 37
4.1 Microarchitecture Independent Characterization . . . 37
4.1.1 Branch Instructions . . . 38
4.1.2 Instruction Footprint . . . 39
4.1.3 Basic Blocks . . . 41
4.1.4 Difference Between Sequential and Parallel Code Sections in HPC Workloads . . . 42
4.2 Microarchitecture Dependent Characterization . . . 43
4.2.1 Branch Predictor . . . 43
4.2.2 Branch Target Buffer . . . 47
4.2.3 Instruction Cache . . . 48
4.3 Impact on Performance, Power and Area . . . 50
4.3.1 Experimental Setup . . . 51
4.3.2 Results . . . 52

5 Sharing the I-cache among Lean Cores 55
5.1 Sequential and Parallel Code within HPC applications . . . 55
5.2 Lean Cores and the Code They Execute . . . 57
5.3 Shared I-cache Architecture . . . 58
5.3.1 Core Front-End . . . 58
5.3.2 Shared I-cache and Interconnect . . . 60
5.4 Simulation Setup . . . 61
5.5 Evaluation . . . 61
5.5.1 Naive I-cache Sharing . . . 62
5.5.2 Scalable I-cache Sharing . . . 64
5.5.3 Miss Analysis . . . 65
5.5.4 Area and Power Savings . . . 66
5.5.5 A Single I-cache Shared Among All Cores on an ACMP . . . 69

6 Multi-socket GPU Design 73
6.1 System of Interest and Simulation Setup . . . 74
6.2 NUMA-Aware GPU Runtime . . . 76
6.2.1 Performance Through Locality . . . 78
6.3 Asymmetric Interconnects . . . 81
6.3.1 Dynamic Bandwidth Distribution . . . 81
6.3.2 Results and Discussion . . . 83
6.4 NUMA-Aware Cache Management . . . 85
6.4.1 Design Considerations . . . 86

6.4.2 Results . . . 89
6.5 Discussion . . . 91
6.5.1 Combined Improvement . . . 91
6.5.2 Scalability . . . 93
6.5.3 Multi-Tenancy on Large GPUs . . . 94
6.5.4 Power Implications . . . 94
6.5.5 Scheduling Improvements . . . 95
6.5.6 Other Asymmetric Link and Cache Partitioning Proposals . . . 96

7 Conclusions 99
7.1 Future Extensions . . . 101
7.2 Work Published . . . 103

Bibliography 105

Abbreviations 105

List of Figures

1.1 High-level overview of a single core pipeline. 3
1.2 Comparison of UMA and NUMA multicore CPUs. 5
1.3 Accelerated computing offloads parallel portions of the application to the accelerator, while the remainder of the code still runs on the CPU. 6
1.4 Heterogeneous node with one CPU and two GPUs. 7
1.5 The OpenMP execution flow. 9
1.6 Thesis timeline. 14

2.1 Different CMP configurations. 20
2.2 Potential speedup obtained by different CMP designs depending on the serial code fraction. 21
2.3 Percentage of workloads that are able to fill future larger GPUs (average number of concurrent thread blocks exceeds number of SMs in the system). 24
2.4 The evolution of GPUs from traditional discrete PCIe devices to single logical, multi-socketed accelerators utilizing a switched interconnect. 25

3.1 Sniper interval simulation model. 28
3.2 Temporal flow of a simulation process with TaskSim. 30

4.1 Dynamic branch instruction breakdown for each benchmark suite as the percentage of total instructions. 38
4.2 Distribution of branch directions. Conditional branches are dominantly decided in one direction, either taken or not taken. Desktop applications have more evenly distributed directions of conditional branches. 39
4.3 Static instruction footprint and memory we need to store 99% of dynamic instructions per benchmark suite. 40
4.4 Average dynamic basic block length and distance between taken branches for each benchmark suite. 41
4.5 Branch MPKI for different branch predictor configurations and benchmark suites. 45

4.6 Branch MPKI breakdown for gshare branch predictor and a subset of workloads. We distinguish mispredictions on not taken, taken backward, and taken forward branches. 46
4.7 BTB MPKI for different number of entries and associativity. We use branch address to index BTB. 48
4.8 The average I-cache MPKI values for all benchmark suites. The cache line is 64 B. 49
4.9 I-cache MPKI values for some specific benchmarks. The cache size is 16 KB. 49
4.10 Normalized execution time, power, energy, and energy-delay (ED) product for different CMP configurations and averaged per benchmark suite. We analyse only cores and L2 caches since the rest of CMP is shared and same for all configurations. Asymmetric++ CMP has the same area budget as Baseline CMP. 51
4.11 Execution time for a subset of benchmarks, normalized to a baseline CMP configuration. 52

5.1 The average dynamic basic block length in serial and parallel parts of the code. 56
5.2 The I-cache MPKI values in serial and parallel parts of the code using a 32 KB, 8-way associative I-cache with 64 B lines, and LRU replacement policy. The I-cache MPKI values in parallel code are very low. 56
5.3 Percentage of instruction sharing across all threads running on an eight-core CMP per HPC benchmark (parallel sections only). 57
5.4 Baseline ACMP architecture with respect to the instruction part of memory hierarchy. 59
5.5 Shared I-cache ACMP architecture. Master core is not modified. . . 60
5.6 Naive scaling. Execution time for different levels of sharing a 32 KB I-cache among worker cores. We use four line buffers and a single bus as the interconnection network. 63
5.7 Naive scaling. Normalized CPI stack per benchmark for the highest level of sharing ($cpc = 8$). 63
5.8 I-cache access ratio for different number of line buffers. More than eight line buffers does not reduce the I-cache access ratio significantly. 64
5.9 Trade-off between adding more line buffers and doubling the interconnection bandwidth when a single 16 KB I-cache is shared ($cpc = 8$). The execution times are normalized to the baseline architecture (private, 32 KB I-caches). 65
5.10 MPKI values for an I-cache shared among all eight lean cores in its two sizes, 32 KB and 16 KB, normalized to a baseline ACMP (private 32 KB I-caches). Numbers above the graph represent absolute MPKI values for each benchmark with private I-caches. 67

5.11 Energy and area savings adding more line buffers and doubling the interconnection bandwidth when a single 16 KB I-cache is shared ($cpc = 8$). All the values are normalized to the baseline architecture and averaged across the benchmarks. 68
5.12 Execution time ratio dependence on the serial code fraction. 69

6.1 Schematic representation of proposed transparent multi-socket GPU system consisting of four GPU sockets and one CPU. 74
6.2 Comparison of round-robin and first-touch allocation policies on a dual-GPU system. 77
6.3 Comparison of traditional and locality optimized CTA scheduling. . 78
6.4 Performance of a 4-socket NUMA GPU relative to a single GPU and a hypothetical 4× larger (all resources scaled) single GPU. Applications shown in grey achieve greater than 99% of performance scaling with SW-only locality optimization. 79
6.5 Example of dynamic link assignment to improve interconnect efficiency. 80
6.6 Normalized link bandwidth profile for `HPC-HPGMG-UVM` showing asymmetric link utilization between GPUs and within a GPU. Vertical black dotted lines indicate kernel launch events. 82
6.7 Relative speedup of the dynamic link adaptivity with respect to the baseline architecture by varying sample time and assuming switch time of 100 cycles. In red, speedup achievable by doubling link bandwidth. 84
6.8 Potential L2 cache organizations to balance capacity between remote and local NUMA memory systems. 85
6.9 Performance of NUMA-aware dynamic cache partitioning in a 4-socket GPU compared to memory-side L2 and previously proposed static partitioning. 89
6.10 How different L2 cache organizations shown on Figure 6.8 affect the execution time in case of `HPC-AMG`. Vertical dotted lines stand for kernel launch events while colors show the number of remote accesses. 90
6.11 Performance overhead of extending current GPU software based coherence into the GPU L2 caches. 91
6.12 Final NUMA-aware GPU performance compared to a single GPU and 4× larger single GPU with scaled resources. 92
6.13 NUMA-aware 1–8 socket GPU scalability compared to hypothetical larger single GPU with scaled resources. 93
6.14 Time line of `HPC-Lulesh` with 10 time-steps executing on a dual-socket GPU. Vertical dotted lines stand for kernel launch events, and color intensity represent the number of remote memory accesses. 95

List of Tables

3.1 Evaluated shared-memory OpenMP benchmarks. 33
3.2 Evaluated CUDA applications with memory footprint and time weighted average number of thread blocks available during execution. 34

4.1 The average percentage of backward and forward taken branches per benchmark suite. 39
4.2 Size parameters and hardware cost of evaluated branch predictors. Parameter n stands for the number of address bits used to index the tables, and parameter m stands for branch history length. . . . 44
4.3 I-cache, BP, and BTB share in total area and power budget on a Cortex-A9 core level. BP has 12-cycle miss penalty. The numbers are obtained using McPAT and CACTI tools with processing technology of 40 nm. 53

5.1 Configuration parameters for the simulated ACMP. 62

6.1 Simulation parameters for evaluation of single and multi-socket GPU systems. 75
6.2 Cache partitioning procedure for NUMA-aware L1 and L2 caches. . 88

Chapter 1

Introduction

1.1 High Performance Computing

Theory and experimentation have stood as vital tools for solving problems and challenges since the beginning of scientific methodology. Today, major breakthroughs in all areas of science and engineering depend on computational approaches. Scientific computing is widely used in situations where solving a problem using traditional scientific methods may be dangerous, even impossible, too expensive, time-consuming, or too complex. Modeling real systems on a computer, researchers are able to gain new insights in fields such as weather prediction, fluid dynamics, stochastic probability, molecular interaction, pattern recognition, new material characterization, machine learning, etc.

In order to solve even larger problems, with less time and higher accuracy, researchers and engineers turn to *High Performance Computing* (HPC). HPC generally refers to a practice of using clusters of computers, called *compute nodes*, that work together running a particular workload, and thus delivering higher performance than a single desktop computer or workstation. Large clusters of computers form a *supercomputer*, a system with many compute nodes connected through a fast interconnection network with access to a distributed and shared storage system. Today's computer technology is driven by the development of supercomputers with architectures and organizations that moved from traditional sequential machines to parallel and distributes systems. Given that most challenging problems

always require more resources than can be provided by the fastest available supercomputer, the need for a system that provides more performance is continuously present [1].

Supercomputers nowdays stand as a serious long-term investment in research and science. The Top500 site [2] maintains a list of the most powerful supercomputers in the world. In general, supercomputers represent large installations with significant cost of deployment and maintenance. Looking at the order of magnitude values, they cost ∼10s of millions US dollars, occupy ∼100s of m^2 in area, consume ∼10s of MW in power, and provide ∼10s of Petaflops performance (10^{15} of floating point operations per second). For example, Europe's fastest supercomputer, named Piz Daint [3], achieves theoretical peak performance of 7.8 Petaflops comprising 5272 compute nodes and drawing 1.3 MW of power. With such a high cost of utilizing and running these systems, the current trend in HPC is not only to increase the performance of future supercomputers but also to improve their energy-efficiency (FLOPS per watt). There is a global effort to build the first capable exascale computing system (performance of an Exaflops) which operates in a power envelope of 20 to 30 MW [4]. To reach this goal, the Piz Daint supercomputer would have to improve its performance by ∼100× but also its FLOPS/W ratio by ∼10×.

HPC applications implement parallel algorithms to make use of systems such as supercomputers. A programmer is responsible for identifying the parallelism and writing the parallel code using explicit constructs provided by the programming model. The most popular and standardized model for distributed memory architectures is Message Passing Interface or MPI [5]. Typically, an application spawns multiple processes, one process per node, decomposing the computational workload and distributing its data across local memories. Communication between processes is achieved through explicit data movement from the address space of one process to that of another through cooperative operations on each process. Interchangeably switching between computation and communication phases, processes iterate over input data, converging to the solution. In a well designed algorithm, most of the time is spent inside computational phases. Improving the performance of a single compute node directly translates to shorter execution time of the entire application. This thesis focuses on optimizing the hardware budget and improving the performance on a compute node level.

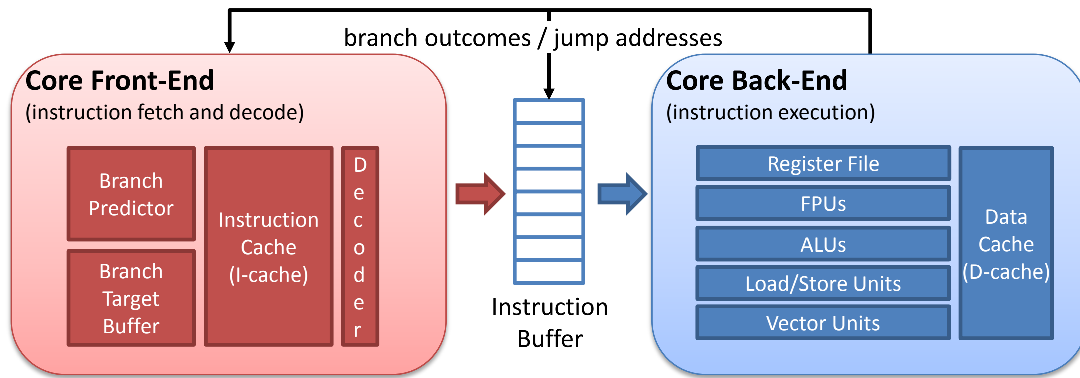


Figure 1.1: High-level overview of a single core pipeline.

1.2 Evolution of a Single Compute Node

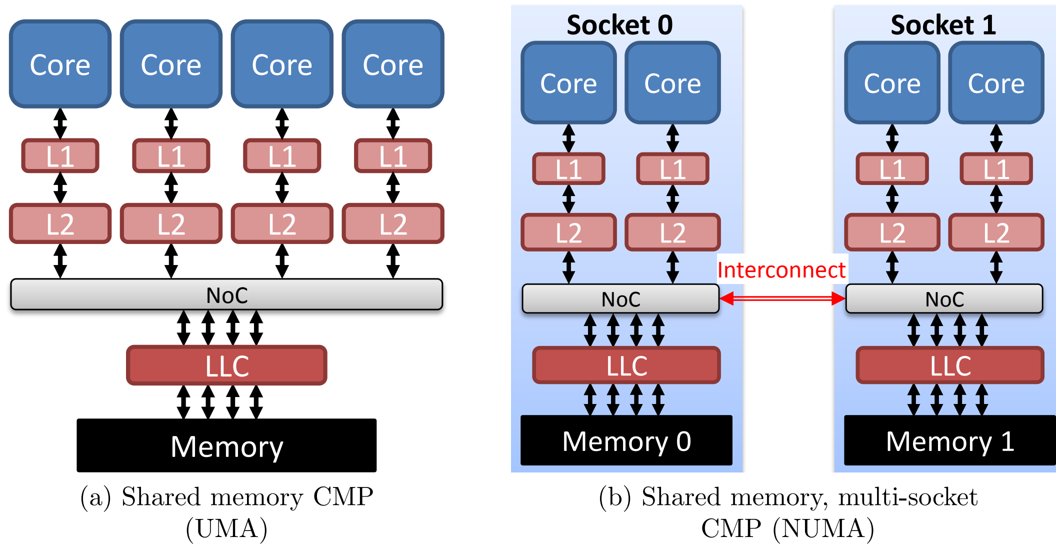
Initially, compute nodes accommodated one or few single-core processors with local main memory, designed for floating point performance [6]. Throughout most of the 80's and 90's, engineers were able to design more capable CPUs by making use of more available transistors on chip, a driving performance force known as Moore's Law [7]. With more transistors, CPUs became more complex trying to optimize the execution flow and do more work per cycle. Cores were able to commit multiple instructions per cycle through out-of-order execution, speculation, predication, on-chip cache memories, data prefetching, etc. As shown in Figure 1.1, general core execution pipeline can be divided into core front-end, responsible for instruction fetching and decoding, and core back-end, in charge of executing and committing instructions. These two mechanisms act as a 'producer' and 'consumer', connected through the instruction buffer. With multiple units tailored for different types of instructions and out-of-order execution, core back-end is capable of consuming more than one instruction per cycle (IPC). To keep these pipeline stages busy, the core front-end also had to be improved with complex branch prediction, instruction prefetching, and decoding more instructions at the time. Many complex features are implemented in today's cores to increase fetch bandwidth, such as multiple branch predictions per cycle [8], instruction alignment [9], branch target buffer [10], trace cache [11], etc. Together with the IPC improvement, as the size of transistors shrunk, power was proportional to the area of the transistors so the entire core could operate at higher frequencies for the same power, a trend known as Dennard's Scaling [12]. Architectural innovations in exploiting instruction level parallelism (ILP) and frequency scaling were the two main sources of continuous improvement of single-core performance.

1.2.1 Multicore Processors

In approximately 2004, processor engineers had reached the transistor size where Dennard's Scaling could not be applied anymore. More transistors were available coming from Moore's Law but it was not possible to keep the power envelope constant anymore. Energy consumption became a new constraint introducing energy-efficiency as a key metric for designs. Thus, major vendors moved to multicore processors giving up on higher clock rates. To exploit this peak performance provided by a multicore CPU, applications had to be rewritten and parallelized by exposing thread level parallelism (TLP).

The first chip multiprocessors (CMP) were symmetric, with the same multiple cores interconnected and attached to the shared local memory. This approach simplified the runtime management and user's view of a system given that a running process can be executed on any of the existing cores providing the same performance. With multiple cores on chip, the memory subsystem had to be improved. A small amount of fast memory, or cache, was added per core, keeping data close to the CPU and available for reuse without extra cycles to store and fetch data from slow main memory. Typically, CMPs today implement several levels of cache hierarchy, with the closest ones to the core being private (L1 and L2), and the last one (LLC) being shared. Furthermore, L1 cache is usually separated, with one part storing instructions (I-cache) and another reserved for data (D-cache). Symmetric CMPs with a single main memory are known as Uniform Memory Access (UMA) CMPs since the memory access time does not depend on the memory location. Figure 1.2a depicts a typical UMA CMP. It shows a four-core CMP where each core has private L1 and L2 caches, with all the L2s connected to the shared LLC and main memory through the network on chip (NoC).

With more cores per CMP, UMA organization became unscalable as the memory capacity per core and the memory bandwidth between cores and main memory could not follow the increase in compute performance. One way to address this issue was by introducing multi-socket nodes with Non-Unified Memory Access (NUMA) CMPs. Cores are organized into sockets each with its own local memory. The memory of other socket is accessible but the latency is increased. Figure 1.2b shows a dual-socket CMP with two cores per each socket. A running thread on `Socket0` can access the data stored in `Memory1` but this remote memory access takes more cycles than accessing local data. Multi-socket NUMA organizations



(a) Shared memory CMP (UMA)

(b) Shared memory, multi-socket CMP (NUMA)

Figure 1.2: Comparison of UMA and NUMA multicore CPUs.

stand as a favorable design increasing the total number of cores, total memory capacity, and memory bandwidth. Still, it introduces a complexity for users and runtime systems. To extract the maximal performance, the number of remote memory accesses has to be reduced, with processes fetching its data mostly from the local memory.

1.2.2 Compute Accelerators

Architects start designing more energy-efficient CMPs with the end of Dennard scaling. This includes workload characterization and appropriate hardware optimization. HPC applications usually consist of sequential and parallel code regions. Cores in a CMP have already been designed to improve scalar performance, with power-hungry pipeline stages able to exploit available ILP. On the other side, parallel code sections in HPC workloads have different characteristics compared to serial code. Relaying on the abundant TLP, and for a given area and power budget, it was more beneficial to implement many low-power cores instead of a few heavyweight ones. According to Amdahl's law [13] and with more cores on a chip and increasing available TLP, the sequential part of the code eventually becomes the bottleneck. To support both serial and parallel code executions, supercomputers start employing accelerators together with the latency-optimized CMPs, forming heterogeneous compute nodes.

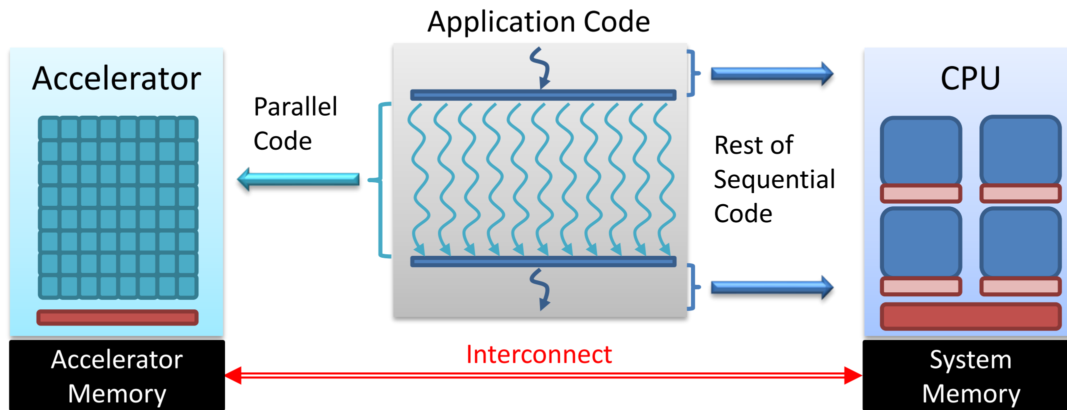


Figure 1.3: Accelerated computing offloads parallel portions of the application to the accelerator, while the remainder of the code still runs on the CPU.

The main idea was to offload the compute intense part of an application to be executed on the accelerator. General overview of the execution flow is shown on Figure 1.3. Accelerators base their compute power and energy-efficiency by exploiting data parallelism, executing the same instructions or tasks on different data portions in parallel. Since the early 2000s, the fastest supercomputers from the Top500 and the most energy-efficient ones from Green500 [14] lists have used various computer accelerators.

ClearSpeed first introduced its CSX600 accelerator designed for HPC applications [15]. It was based on a Single Instruction Multiple Data (SIMD) execution model, with an array of 96 processing elements running under 25 W of power envelope. At that time, rewriting HPC code for a specific hardware was not a popular practice, and manufacturing an HPC accelerator was not profitable for the company. Luckily, video and gaming industry was driving the development of data parallel accelerators. IBM presented its Asymmetric CMP (ACMP) design with the Cell/B.E. and PowerXCell architectures [16] implementing one big and 8 energy-efficient cores. In 2008, supercomputer Roadrunner [17] was ranked first on Top500 and fourth on Green500 list, with 12960 IBM PowerXCell and 6480 AMD Opteron dual-core processors. Following the path, Intel developed its Larrabee [18] and Xeon Phi [19] throughput-oriented CMP with many lean cores targeting both graphics and HPC workloads. NVIDIA and AMD offered their graphical processing units (GPUs) [20] originally designed for multi-dimensional rendering in video games. Today, HPC-optimized GPUs implement massively parallel architectures with thousands of processing units reaching peak performance of ∼10 Teraflops

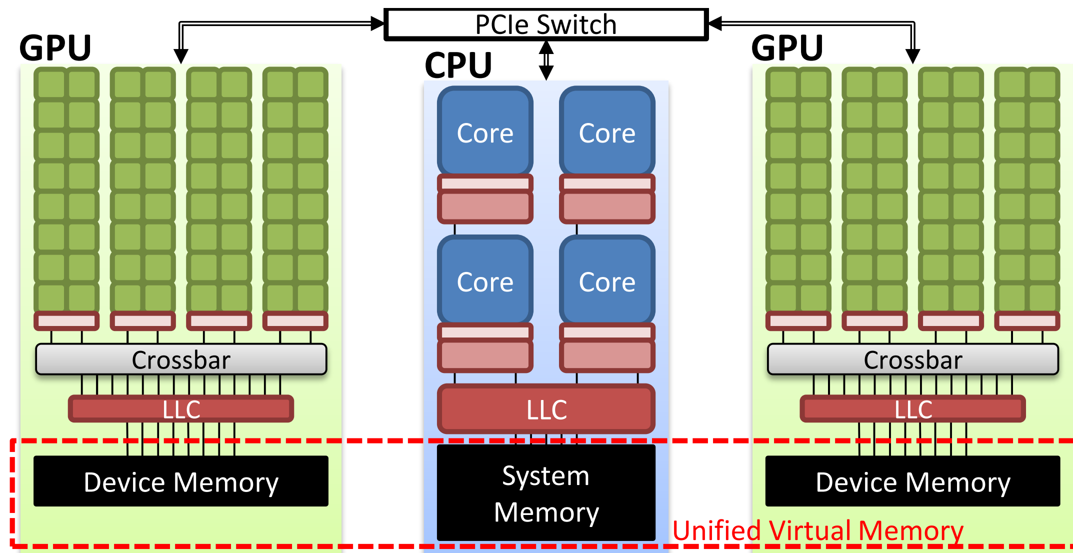


Figure 1.4: Heterogeneous node with one CPU and two GPUs.

and memory bandwidths of ∼1 TB/s making them particularly suitable for data parallel HPC workloads [21].

1.2.3 Multiple Compute Accelerators on a Single Node

With the increased amount of data to be processed, HPC workloads contain sufficient data parallelism to fill accelerators that are 2–8× larger than today's biggest GPU. For that reason, current systems implement nodes with one CMP optimized for serial and multiple devices (many-core accelerators or GPU) for parallel execution, interconnected through PCIe links or some other proprietary low-latency, high-bandwidth interconnect. At the moment, the most energy-efficient supercomputer on Green500 list is NVIDIA's DGX SATURNV, made out of compute nodes with 8 GPUs, for 9462 Megaflops/W. Figure 1.4 presents the example of a heterogeneous compute node with two GPUs deployed as accelerators. The CPU is optimized for single-thread performance and latency, with several levels of cache hierarchy, large control units, large main memory, and low memory bandwidth. Contrarily, the GPUs are designed for parallel performance and throughput, with many simple floating-point and integer units, simplified control logic, and small caches. To provide enough data to this increased amount of processing elements, GPUs have high-bandwidth main memory, typically smaller in capacity compared to the one in CPUs. With multiple physical memories available on a node, the problem of optimal data placement still exists, just as for multi-socket NUMA

CMPs. Current runtime systems lessen the burden on the programmer by allowing different memories to appear as a single unified memory space. Code complexity is reduced since explicit data movement is no longer required. However, to gain the most of performance, developers need to optimize the application by providing data placement hints to the compiler and runtime.

In this thesis, we try to optimize the compute nodes for parallel code execution. The focus is on reducing the execution time as well as power and area consumption through better utilization of available transistors on chip. For many-core accelerators we evaluate the idea of sharing the first level instruction cache (I-cache) among multiple low-power cores. For the applications that provide parallelism even beyond a single GPU accelerator, we study the architectural improvements needed for efficient and transparent multi-GPU executions.

1.3 Programming Models for Single Node Architectures

The variety of single node architectures comes with the same variety of programming models. Every time architects introduce performance improvement by exploiting new parallelism, applications had to be rewritten. The vector era of the 1970s and 1980s brought code vectorization in order to exploit vector processors. When commodity scalar processors connected by a network proved to be more cost-effective than vector machines, algorithms were again redesigned for parallel programming using MPI. With CMPs, the next boost in performance came from parallelization across multiple cores sharing a single physical memory. Each MPI process had to be further parallelized spawning multiple threads, typically one thread per core. With the rise of NUMA systems, programming models had to provide the interface to the users for optimal data placement, process and thread scheduling, synchronization, use of specific on-chip memories, etc. Heterogeneous nodes introduced yet another code adaptation, with parallel code being executed on different instruction set architecture (CPU vs GPU) thus requiring different set of compilers and libraries. Code rewriting and tunning for new architectures and programming models stand as a serious decision for HPC users.

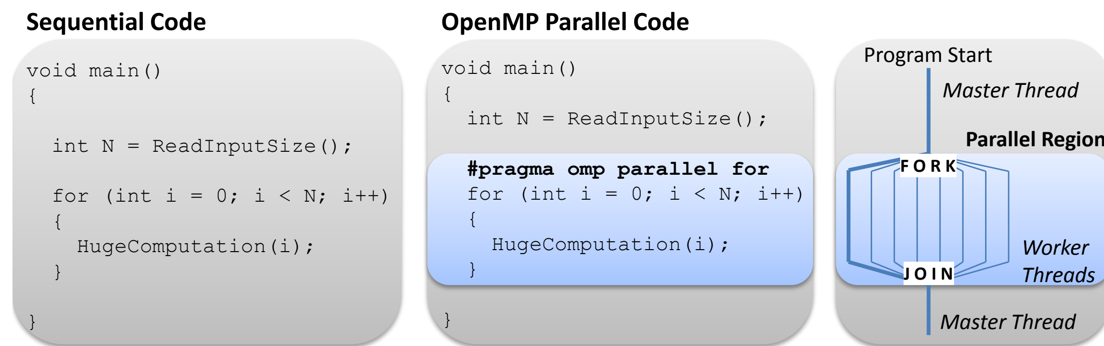


Figure 1.5: The OpenMP execution flow.

1.3.1 OpenMP for Shared Memory Multicore Processors

As multicore processors emerged, there was a need to provide a portable, standardized, and scalable programming model for users to parallelize their applications. OpenMP [22] stands as an interface, specified for C/C++ and Fortran programming languages, that provides a way for the sequential programs to expose their thread based parallelism. It is designed for multicore, shared memory UMA or NUMA machines. The main idea is to identify the most time consuming parts of a workload, typically loops, and distribute the loop iterations across multiple threads by inserting compiler directives. OpenMP uses the *fork-join* model of parallel execution. An application starts as a sequential program run by the *master thread*. At the moment it reaches a directive for creating a parallel region (`#pragma omp parallel`), the master thread creates a team of *worker threads* now running in parallel (fork), usually one thread per core. Upon termination, all threads synchronize, leave the parallel region with only the master thread to continue the execution (join). Worker threads do not maintain the exact consistency thus it is a programmer responsibility to ensure the correct update of shared variables by using critical sections, semaphores, or atomic operations. Figure 1.5 compares the sequential and OpenMP version of the same code section. In case of serial code, a single thread iterates `N` times and performs some long computation for each iteration. Just by inserting `#pragma omp parallel for` directive, OpenMP version spawns multiple worker threads, running them in parallel. Each thread takes only a subset of iterations in this parallel loop. For example, if we create 8 threads inside the parallel region, `thread0` computes only for `i` in `[0, N/8)`, `thread1` for `i` in `[N/8, N/4)`, etc.

Parallelizing an application using OpenMP may be as simple as inserting one-line compiler directives inside a serial version, but it also provides a rich set of calls for more performance tuning. For example, by setting different environment variables, an advanced user can impose better loop distribution if the default one provides a significant load imbalance. Some applications may prefer to run neighboring threads on the same socket to make use of constructive data prefetching in the shared memory structures (main memory or LLC). To exploit this program behavior, an OpenMP runtime may define proper thread to core binding. Recently, OpenMP added new constructs to support task-based parallelism and efficient execution on vector units [23]. With the appearance of accelerators, runtime now supports the offload of parallel regions to another device, GPU or many-core CMP. Due to its simplicity and support from major hardware and software vendors, OpenMP is today a standard programming model for parallelism on a node level in HPC. In this thesis, we evaluate OpenMP applications running on HPC systems, find some intrinsic properties, and propose better hardware utilization on a CMP level to increase the performance per power and area unit.

1.3.2 OpenACC for Programming Accelerators

OpenACC [24] is short for Open Accelerators, the effort of applying directive-based programming to compute accelerators. The idea of OpenACC is similar to OpenMP, with the extension that compiler hints should be portable across accelerators of various kinds. The main advantage of this programming model is performance portability. The most popular accelerators (many-core devices from Intel and ARM, GPUs, FPGAs) have different parallelism profiles, different amounts of SIMD parallelism, number of cores, and different sorts of sharing inside the memory hierarchy. In order to extract performance from a single code version, OpenACC provides flexibility in how that parallelism gets mapped to the target hardware.

Still, with portability comes less performance. The advanced users tend to narrow the gap between peak theoretical and achieved performance, turning to code redesigning and tuning.

1.3.3 CUDA for Programming GPUs

Developed by NVIDIA, the *Compute Unified Device Architecture* (CUDA) programming model [25] allows a user to program GPUs as compute accelerators where sequential code runs on the CPU (also known as *host*) and parallel code runs on one or more GPUs (*devices*). An application written with CUDA is launched on the host, manages physical memories on both the host and device, and invokes *kernels* which are parallel code sections executed on the device. These kernels are run by many GPU threads grouped in *thread blocks* or cooperative thread arrays (*CTAs*). Each block is executed by one SM and does not migrate, while the number of concurrent blocks that can reside on SM depends on available resources (register and on-chip memory usage). Threads within a CTA can synchronize and communicate, while different CTAs can not, and the program correctness should not depend on the order of CTA execution. Different threads and blocks can be distinguished by their unique ID making them easier to identify and distribute the workload. Blocks make a *grid* which presents a top level abstraction of thread hierarchy. The number of threads per block and the number of blocks in a grid can be set at runtime for each kernel launch. The device itself is consisted of *streaming multiprocessors* (SM) so that each block of threads is running on one of them. Threads within a block are grouped into *warps* as a basic unit of scheduling. All threads in a warp execute the same instruction since they share same program counter but operate on different data. This design point is similar to vector units in CPUs, where initial instruction fetching and decoding is the same for all threads in a warp, thus doing it once per warp instead of once per thread saves the energy.

Communication and data transfers between the host and CUDA device is done using global memory. It is the largest but also the slowest one in memory hierarchy on a GPU. Each thread from any block can both read and write to global memory. Beside that, each SM has small programmable shared memory and/or non-programmable D-cache. Proper and extensive use of on-chip shared memory can significantly increase performance of kernels. In GPUs, a prolonged memory access causes a current running warp to stall. The warp scheduler on a SM then picks some other warp that has ready operands for its next instruction. With a huge number of threads running, SMs in a GPU switch between warps hiding the memory latency. This Single Instruction Multiple Threads (SIMT) execution is similar to fine-grained multithreading on CPUs [26, 27], while threads in a warp

execute the instructions like vector units [28]. To provide data to all the running threads, memory hierarchy is designed for bandwidth. Per-SM cache structures filter the memory accesses going to global memory and coalesce potential cache misses.

Until CUDA version 4, users had to explicitly allocate memory on host or device and copy data before they launch a kernel to run on a GPU device. CUDA has supported Unified Virtual Addressing (UVA) since CUDA 4, which provide a single virtual memory address space for all physical memories in the system. This feature simplified the user code complexity by allowing `mem-copy` calls to be used without exactly knowing where data resides. Still, UVA does not automatically migrate data from one physical location to another. CUDA 6 [29] introduced a concept of Unified Virtual Memory (UVM) where the runtime automatically migrates data from one physical memory to another (CPU or GPU) making any data accessible to the both CPU and GPU(s) using a single pointer. CUDA 8 [30] further improved UVM by enabling on-demand page migration and using system-wide atomic operations. Our optimizations in multi-GPU compute nodes are based on these features.

Extracting the maximal performance from a GPU device requires significant application tuning and knowledge of both programming model and underlying hardware organization. Attractive with the theoretical peak performance they provide, GPUs are the most commonly used accelerators in HPC today. As stated above, the current trend for supercomputers is to have multiple GPU devices per compute node. To utilize them, yet another code rewriting is needed, typically partitioning serial code into multiple contexts (MPI processes or OpenMP threads), one context per GPU. Parallel code is again enclosed into kernels, now launching them on every GPU. This thesis explores the architectural enhancements needed to support efficient and transparent execution on multi-GPU systems.

1.4 Thesis Contributions

The main objective of this thesis is to optimize the hardware budget and improve the performance of compute nodes used in HPC. The following are summarized

novel contributions of this thesis, while we provide more details in the following chapters.

- We first focus on the HPC code characteristics and core front-end which factors around 30% of core power and area on the emerging lean-core type of processors used in HPC. Separating serial from parallel code sections inside applications, we characterize HPC benchmarks and compare them to a traditional set of desktop integer workloads. HPC applications have biased and mostly backward taken branches, small dynamic instruction footprints, and long basic blocks. Our findings suggest smaller branch predictors (BP) with the additional loop BP, smaller branch target buffers (BTB), and smaller I-caches with wider lines. The difference between serial and parallel code sections in HPC applications points to an ACMP design, with one baseline core for sequential and many HPC-tailored cores designed for parallel code. Without performance degradation, we demonstrate potential power and area saving by 7% and 16% respectively, by avoiding over and under provisioning of hardware resources.
- With the previous findings, we also detect that parallel threads in HPC applications execute the same code at approximately the same time. This makes sharing the core front-end structures a potentially beneficial solution. We explore further tailoring of an ACMP design for HPC by sharing a smaller I-cache among worker cores. Our analysis of the multiple parameters finds the sweet spot on a wide interconnect to access the shared I-cache and the inclusion of a few line buffers to provide the required bandwidth and latency to sustain performance. The evaluation on a rich set of HPC benchmarks shows 11% area saving with 5% energy reduction at no performance cost.
- With more lean cores available on chip, we are entering the area of accelerators and GPUs. Most of the single-GPU optimized workload already contain sufficient data parallelism to fill even larger GPUs than we have today. Transparent workload scaling seems attractive from the programmers perspective. We examine the performance of a future multi-socket GPU to understand the effects that NUMA will have when executing applications designed form UMA GPUs. Optimizing GPU microarchitecture for NUMA-awareness is essential to preserve data locality and reduce the inter-socket

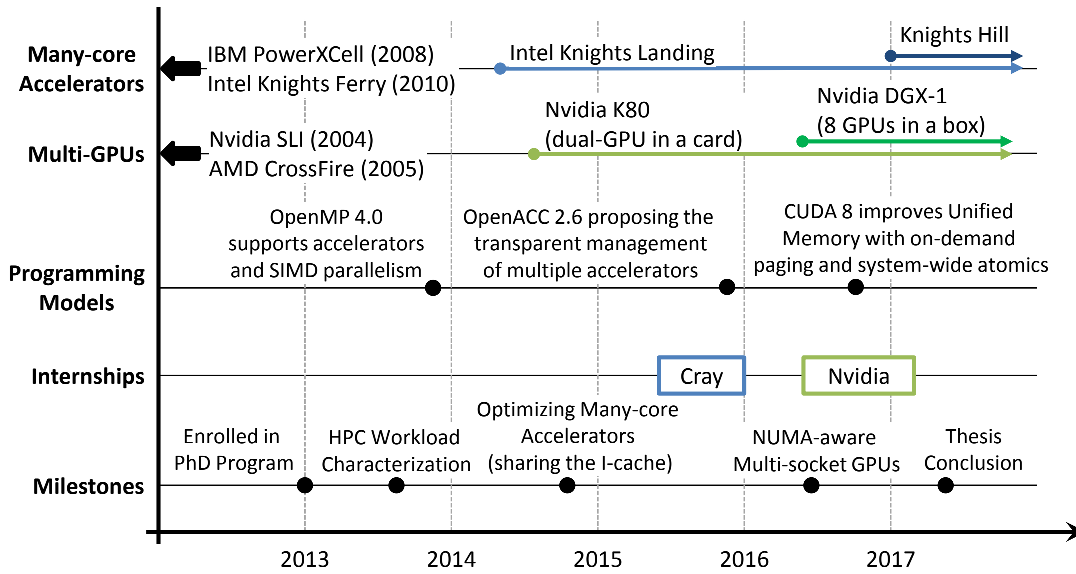


Figure 1.6: Thesis timeline.

bandwidth. To overcome this bottleneck we propose two classes of improvements. First, we show that inter-socket links should be dynamically and adaptively reconfigured at runtime to maximize link utilization. Second, we propose that GPU caches should be made NUMA-aware and dynamically adapt their caching policy to minimize NUMA effects. Our NUMA-aware GPU outperforms a single GPU by 1.5×, 2.3×, and 3.2× while achieving 89%, 84%, and 76% of theoretical application scalability in 2, 4, and 8 sockets designs respectively. We show that multi-socket NUMA-aware GPUs can allow traditional GPU programs to scale efficiently, providing significant room before developers must re-architect applications to obtain additional performance.

1.5 Timeline

Figure 1.6 gives an overview of the major thesis milestones, within a context of related events coming from the industry. By the time work on this thesis had started, IBM with PowerXCell and Intel with Knights Ferry already have introduced compute accelerators for HPC. OpenMP Advisory Board followed up with support for compute offloading to accelerators and SIMD parallelism for vector units. As Intel continued to improve the single-thread performance of their cores

in Knights Landing, in this thesis we improve the utilization of hardware budget by sharing the I-cache. Multiple accelerators per compute node were already entering the market at that time, like in an example of NVIDIA introducing the K80 as a dual-GPU card, although, multi-GPU solutions were known to improve graphics performance (NVIDIA SLI and AMD CrossFire). By the end of 2015, OpenACC proposes transparent management of multiple accelerators as a feature in their upcoming 2.6 version of specification. Finally, NVIDIA enhance the CUDA to provide Unified Memory with on-demand paging and system-wide atomics. Based on these features, we evaluate NUMA-aware multi-GPU systems as a way of improving the scalability in compute nodes with multiple GPU devices.

The course of this thesis was backed by two industrial internships. The first one at the Cray Research Center (Bristol, UK) for performance analysis and evaluation of multi-core and many-core HPC systems was done from September to December 2015. The second internship was at NVIDIA (Santa Clara, USA) from August to December 2016. Together with the beginning of 2017, it was focused on enabling and improving transparent multi-GPU execution in HPC.

1.6 Thesis Organization

The rest of the thesis is organized as follows. The next chapter presents an overview of motivational facts, background, and related work. After that, Chapter 3 provides an overview of methodology we were using throughout of this thesis, describing simulation frameworks and evaluated benchmarks. Chapters 4, 5, and 6 present the results and discuss the contributions of this thesis. Finally, Chapter 7 concludes this work and provides some potential directions of future work.

Chapter 2

Background and Related Work

2.1 From General-purpose to Specialized Systems-on-Chip

With more transistors per chip coming and single-thread performance reaching its plateau, architects moved to specialized core design to suit particular set of applications. In the mobile industry, ARM's low-power cores [31, 32] have been designed to provide long battery life and efficient support for typical mix of mobile workloads. Although still behind Intel's and IBM's general-purpose cores in terms of brute performance, there is an effort to put ARM's licensed CPUs inside data-center and HPC systems [33, 34], thanks to their better energy-efficiency. Deep learning applications now rapidly gain attraction, with some companies finding it cost-effective to manufacture accelerators for this type of workloads [35]. Gaming industry develops their own Systems-on-Chip (SoC) giving more realistic experience to the users consuming video-interactive context [36, 37]. Understanding the workload characteristics is important in the design of efficient computer architectures. Focusing on sequential execution, researchers have evaluated desktop SPEC CPU [38] and other server suites to analyze potential improvements for next-generation chip designs [39–41]. In the first part of this thesis, we focus on HPC workload characterization, with the goal of tuning today's CMP microarchitecture to suit the particular features and requirements of HPC applications.

Today's HPC compute nodes are made of CMPs tailored for desktop and server applications. They usually have few heavyweight cores capable of exploiting the available ILP through wide super-scalar out-of-order execution. The core front-end is designed to support large instruction footprints and to predict the outcomes of branches in complex control flows. On the other side, HPC workloads are different, running in parallel, thus demanding throughput-oriented CMPs. Keeping the same power and area budget, a handful of heavy cores are replaced by many lean ones, integrated as an alternative to exploit TLP and data parallelism. For example, Intel's Xeon Phi and IBM's BlueGene/Q [42] CMP architectures integrate many power-efficient lean cores targeting parallel HPC workloads. With the current configuration of the front-end structures, an embedded processor spends 42% of its energy on instruction supply [43]. Instruction fetches and branch predictions consume 30% of the total power in the ARM Cortex-A15 core [44]. McPAT tool for power and area estimations shows that lean cores, such as ARM's Cortex-A9 and Sun's Niagara2, spend 25% of the total core area, and 30% of the total core power on instruction delivery [45]. Therefore, it is important to evaluate microarchitectural optimizations to lessen front-end activity and area which can have significant impacts on overall power consumption.

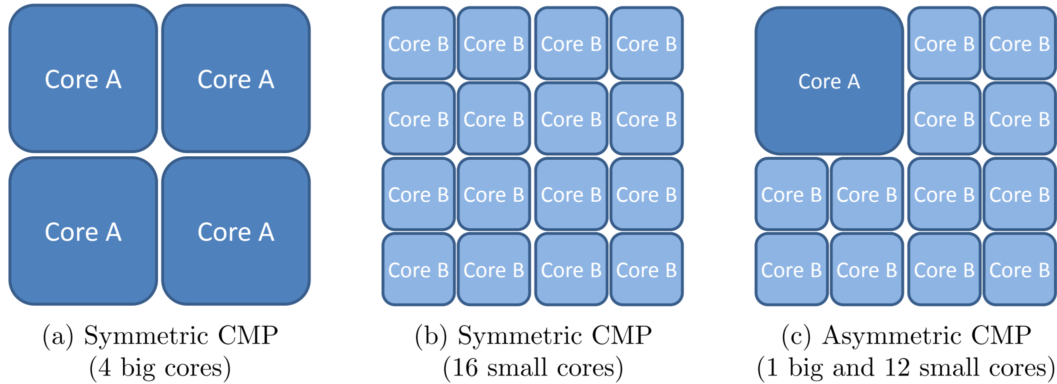
There is a broad scope of previous application performance analyses run in HPC systems. Most of them evaluate inter-node communication overheads, scalability, bandwidth requirements, and data access behavior [46–48]. Exploiting the available data parallelism, vector processors became particularly useful for HPC applications. A single vector instruction can replace an entire loop, and so the instruction fetch and decode bandwidth needed to keep multiple functional units busy is reduced. Many aspects of the analysis performed in this thesis was done in the past in the context of vector machines [49–51]. Scientific and engineering applications have small instruction footprints, long basic blocks, and low control divergence which makes them suitable for SIMD execution. Nowadays, Intel's Xeon Phi cores [52] or Fujitsu's SPARC64 series of chips [53] implement wide vector units to exploit these code characteristics and gain performance. Our work here revisits these findings considering modern HPC workloads and in the context of CMPs and accelerators made out of lean cores.

GPU devices used as accelerators in HPC systems have completely redesigned

front-end compared to traditional CPUs. Power-hungry branch prediction structures are not implemented, and pipeline stalls caused by prolonged branch resolution are leveraged by running many threads concurrently. A programmer has to be aware of reducing the control divergence among threads in a warp, since they all execute the same instructions at any given cycle. Recent study shows that about 95% of branches executed on GPUs can be correctly predicted either with a bimodal or a branch predictor based on local history tables [54]. The cache hierarchy that services the instruction supply is finely tuned for HPC applications. The first-level I-cache is small (4 KB) with wide cache lines (256 B) [55]. All of the scalar cores in a single SM share one I-cache, with threads fetching and executing the same instruction in lock-step mode every cycle [56]. We show that similar front-end organization should be designed for future CMPs used in HPC, allowing each core to run its thread independently.

Focusing on the microarchitectural changes, there are examples where commodity CPUs have been redesigned to suit better an application domain. A recent study calls for a change in future core design identifying the key micro-architectural needs for emerging scale-out workloads as the opposite of traditional scale-up applications used in data centres [57]. Server applications have large instruction footprints and most stall cycles come from I-cache misses [41]. Because of that, ARM's Cortex-A57 cores, used in micro-servers, have a larger 48 KB I-cache to reduce the impact of I-cache misses [58]. An Intel Xeon Phi core has 512-bit wide vector processing unit so it can exploit the SIMD characteristics of scientific codes [59]. Our findings suggest that a similar core tailoring can be applied to lean-core CMPs used in HPC by redimensioning the existing structures based on application demands.

The first contribution of this thesis covers the HPC workload characterization focusing on the core front-end. We evaluate several OpenMP benchmark suites and compare them with traditional desktop applications, found in SPEC CPU INT. We analyze architecture independent code properties, followed by architecture dependent implications. Our results show that HPC applications expose different code characteristics, quantifying each of them. With those findings, we give recommendations on how to adequately dimension the core front-end structures of lean cores for HPC workloads to get maximal area and power savings without performance impact. Moreover, we analyze the difference between serial and parallel



(a) Symmetric CMP (4 big cores)
(b) Symmetric CMP (16 small cores)
(c) Asymmetric CMP (1 big and 12 small cores)

Figure 2.1: Different CMP configurations.

code sections inside HPC workloads. Not just that HPC cores should be tailored differently from desktop, but also the master core has to be tailored differently from worker cores.

2.2 Efficient CMP Design for HPC

As we have mentioned earlier, HPC applications run both in sequential and parallel manner. In case of OpenMP programming model, master thread running on a master core executes the serial code sections and joins the worker threads, which run on worker cores, to execute parallel code regions. To reach an efficient CMP design we need to understand both serial and parallel code in order to improve the utilization of available transistors avoiding over and under provisioning.

On a CMP level, previous work suggested an Asymmetric CMP (ACMP) design, where multiple single-ISA cores exist on a chip, but with different power, area, and performance characteristics [60]. Let's assume there is a CMP hardware budget (whether it is power, area, or some combination of different factors) equal to 16 *core units*. Next, we can consider Pollack's rule [61] and assume that core performance increase is proportional to square root of increase in complexity. In other words, if we quadruple the hardware budget in a core, it will deliver 2× more performance. The question is how to distribute the total hardware budget among multiple cores to efficiently execute both serial and paralle code regions in HPC?

Figure 2.1 shows three possible solutions. First two, shown on Figures 2.1a and 2.1b, present Symmetric CMPs (SCMP) with all the cores on chip being

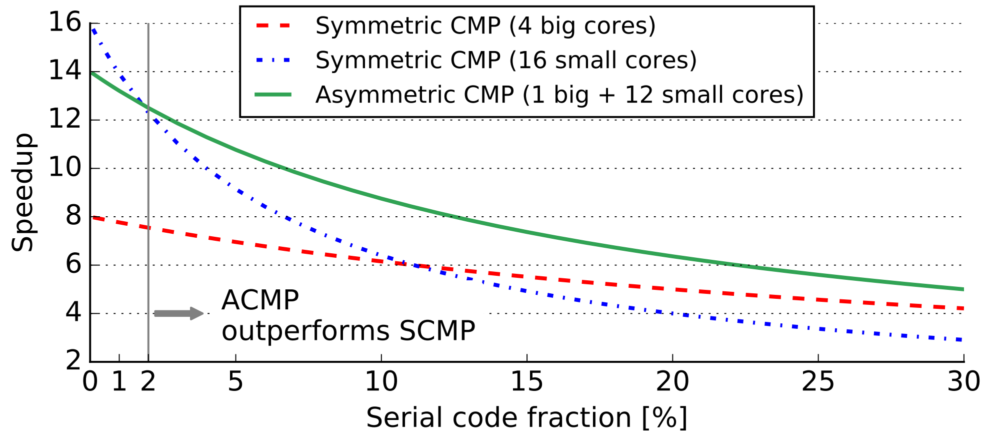


Figure 2.2: Potential speedup obtained by different CMP designs depending on the serial code fraction.

the same in performance. The total hardware budget can be spent either by implementing four big cores (*CoreA*), each providing 2× more performance, or by implementing 16 small cores (*CoreB*), each with 1× performance. Another option is Asymmetric CMP (ACMP) design, with one big core for executing sequential code, and the rest of small cores for parallel code.

Figure 2.2 shows the potential speedup that different CMP designs can provide depending on the serial code fraction for a parallel workload. An ACMP outperforms both symmetric CMP designs when the application has more than 2% of serial code fraction. As the number of cores on a chip increases, the amount of time spent inside the serial code becomes larger. An ACMP design stands out as a solution capable of efficiently executing both parallel and sequential code regions, combining a latency-oriented core with a set of throughput-oriented cores. In this thesis, we evaluate an ACMP design with one big and eight lean cores. We note that the throughput-designed cores execute the same code at approximately the same time, thus analyze the idea of sharing a single I-cache among these cores. We show that such cache organization does not hurt the performance but saves power and energy.

As soon as we start sharing resources among cores in a CMP, we enter the blurred space between multicore and multithreaded processors. The first papers dealing with simultaneous multithreaded (SMT) processors already identified the shared front-end as one of the major bottlenecks [62]. There have been proposals and products for multithreaded processors with a lower resource sharing degree than

SMT. Conjoined cores [63], CASH [64], IBM Cyclops64 [65], and AMD's Bulldozer module [66] propose a CMP where adjacent cores share some of the hardware structures such as the I-cache, the data cache, and the floating point unit.

All of the previous proposals focus on sharing resources among two adjacent heavy-weight cores, while our intention is to provide a thorough analysis on sharing only the I-cache among many worker cores on an ACMP. Since the rest of the core front-end is not shared, this design improves scaling and it allows sharing among more than two cores. Our work points the limiting factors with more cores sharing an I-cache, with the main objective of increasing performance for the same hardware budget.

The I-cache sharing has also been studied for OLTP workloads [67], which have instruction footprints that exceed the capacity of the I-cache in general-purpose processors. Their design advocates for sharing a larger capacity I-cache to reduce the number of misses in the I-cache. We show that a single shared I-cache, smaller than a private one, reduces the number of I-cache misses due to inter-thread prefetching, and also leads to area (and power) savings. In their work, the authors focus only on miss analysis not concerning the implication of the proposed design on execution time, as we do here.

Sharing the I-cache among many low-power embedded processors has also been evaluated [68]. Their work is focused on embedded micro-kernels and caches of 1 KB in size. They observe performance improvements up to 60 %, and identify conflicting accesses to the shared I-cache as a potential source of problems. In this paper, we evaluate mechanisms to hide the extra latency involved in conflicting accesses to the shared I-cache and interconnect.

Finally, in the context of HPC workloads, NVIDIA GPU accelerators [69] already use a shared I-cache for all CUDA cores in a SM. Threads in a warp fetch and execute the same instruction in lock-step mode every cycle, which prevents conflicting I-cache accesses and latency variations. We evaluate this approach in a more general way focusing on ACMPs where each thread has its own program counter and executes a separate instruction stream without any constraints.

Decoupling latency-optimized core(s) from throughput-optimized ones leads to a heterogeneous compute node. For serial code, nodes today use general-purpose CPUs while parallel code is offloaded to an accelerator, a many-core CMP or a

GPU. Our shared I-cache proposal can be applied to many-core accelerators such as Intel's Xeon Phi. With hardware savings and without performance degradation, this contribution points one way of increasing the performance per area and power unit, the ultimate goal of energy-efficient computing.

2.3 More Performance per Compute Node

However, if we are interested in solely scaling the performance, we need to explore the other ways of having more capable compute nodes. Particularly looking at the parallel code sections, the question is can we make it run faster. This means either improving the single-thread performance for each lean core in an accelerator, or increasing the throughput by putting more cores. The first option balances between having more simple cores and less beefier ones. Intel Xeon Phi CMPs are following this path, with the latest Knights Landing cores providing 3× more single-thread performance than the previous, Knights Corner generation [70]. The second option seems to be pursued by GPU manufacturers, increasing the number of SMs per device from one generation to another. This performance scaling was based on Moore's Law and significant growth in per-GPU transistor count and main memory DRAM bandwidth. For example, NVIDIA's GPUs based on Fermi architecture integrated 1.95B transistors on a 529 mm^2 die, with only 16 SMs [71], while the latest Pascal architecture contained 12B transistors within 610 mm^2 die, having 56 SMs. Unfortunately, transistor density is slowing significantly and many integrated circuit manufacturers are not providing roadmaps beyond 7 nm. Moreover, GPU die sizes, which have been also slowly but steadily growing generationally, are expected to slow down due to limitations in lithography and manufacturing cost.

Building larger GPUs with more SMs is a laudable goal, but we must first understand if today's single-GPU applications have enough parallelism to exploit them. Today NVIDIA's largest GPUs contain 56 SMs; across a benchmark set of 41 applications (later described in Section 3.3.2), Figure 2.3 shows that most single GPU optimized workloads already contain sufficient data parallelism to fill GPUs that are 2–8× larger than today's biggest GPUs. While these applications

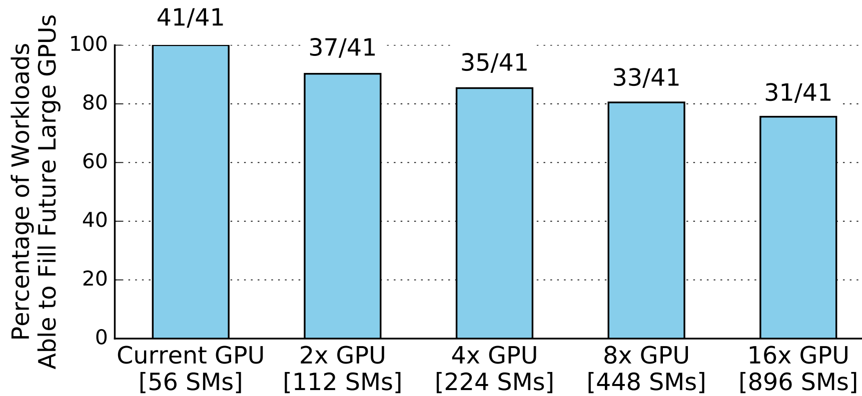


Figure 2.3: Percentage of workloads that are able to fill future larger GPUs (average number of concurrent thread blocks exceeds number of SMs in the system).

are unlikely to ever scale to tens of thousands of GPUs across an entire data center, we believe that programmer transparent workload scaling (up to 8×) will be attractive to many GPU developers.

Without either larger or denser dies, GPU architects must turn to alternative solutions to significantly increase GPU performance. Recently 3D die-stacking has seen significant interest due to its successes with high bandwidth DRAM [72]. Unfortunately 3D die-stacking still has a significant engineering challenges related to power delivery, energy density, and cooling [73] when employed in power hungry, maximal die-sized chips such as GPUs. Thus we propose GPU manufacturers are likely to re-examine a tried and trued solution from CPU world, *multi-socket GPUs*, to scaling GPU performance while maintaining the current ratio of FLOPS and DRAM bandwidth.

Multi-socket GPUs are enabled by the evolution of GPUs from external peripherals to central computing components, considered at system design time. GPU optimized systems now employ custom package designs that accommodate high pin count socketed GPU modules [74] with inter-GPU interconnects resembling NVLink, QPI or HyperTransport [21, 75, 76], as shown in Figure 2.4. Additionally, high port-count PCIe switches are now readily available and the PCI-SIG roadmap is projecting PCIe 5.0 bandwidth to reach 128 GB/s in 2019 [77]. The expansion of GPUs from single pluggable devices to closely coupled multi-socket designs is a natural progression towards exploiting available data parallelism. These multi-GPU

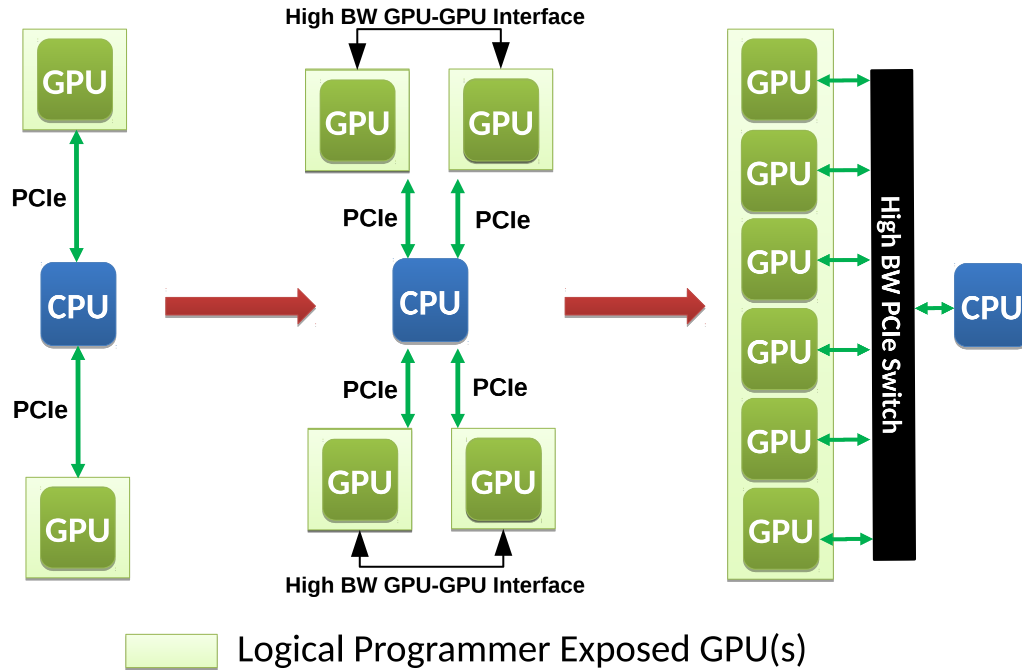


Figure 2.4: The evolution of GPUs from traditional discrete PCIe devices to single logical, multi-socketed accelerators utilizing a switched interconnect.

systems can provide high aggregate throughput when running multiple concurrent GPU kernels, but to accelerate a single GPU workload they require layering additional software runtimes on top of native GPU programming interfaces such as CUDA or OpenCL [25, 78]. Unfortunately, by requiring application re-design many workloads are never ported to take advantage of multiple GPUs.

Several groups have previously examined aggregating multiple GPUs together under a single programming model [79, 80]; however this work was done in an era where GPUs had limited memory addressability and relied on high latency, low bandwidth PCIe interconnects. As a result, prior work focused primarily on improving the multi-GPU programming experience rather than achieving highly scalable performance. Building upon this work, we propose a multi-socket *NUMA-aware* GPU architecture and runtime that aggregates multiple GPUs into a single programmer transparent logical GPU. We show that in the era of unified virtual addressing [29] and cache line addressable high bandwidth interconnects [21], scalable multi-GPU performance may be achievable under existing single GPU programming models.

Next three Chapters stand for the three main contributions of the thesis. We first provide a detail HPC workload characterization targeting core front-end structures. Next, we evaluate the idea of sharing the I-cache among multiple lean cores in an ACMP or many-core accelerator. The third chapter tackles the challenges in further performance scaling of compute nodes with multiple GPU devices as accelerators.

Chapter 3

Methodology

3.1 Code Instrumentation

We use Pin [81] as a tool for dynamic instrumentation of application binaries. It provides an infrastructure for writing program analysis tools called *PinTools*. In a PinTool, the user defines instrumentation routines and what characteristics of an application to instrument. Then, at runtime, those instrumentation routines insert calls to user defined analysis methods where inspection and data collection is performed. The workload runs on top of Pin, which captures relevant information from the workload and passes it to the pintool.

In case of architecture independent characterization, they only collect the statistic in their analysis routines. For example, in the case of branch instruction analysis, a PinTool counts the number of branches, checks for each one if it is taken or not taken, etc. For architecture dependent characterization, PinTools simulate specific HW structures. In the case of instruction fetching, a PinTool configures an I-cache and instruments each instruction analyzing if its address hits or misses in the I-cache.

Fast and simple, this instrumentation library provides a common tool for workload characterization. In this thesis, we implement and use different PinTools to analyze code behavior regarding core front-end structures such as I-cache, branch predictor (BP), and branch target buffer (BTB). Still, for performance and power modeling, we switch to more accurate and validated simulation frameworks.

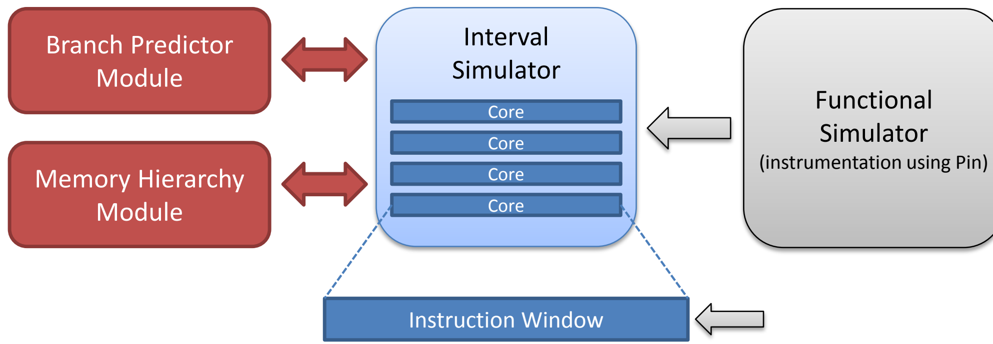


Figure 3.1: Sniper interval simulation model.

3.2 Simulation Frameworks

3.2.1 Sniper

Sniper [82] is an event-driven multi-core simulator based on interval simulation. It trades simulation accuracy for speed in order to provide an infrastructure for design space exploration on the level of homogeneous and heterogeneous CMPs. The main idea is to use analytic and mechanistic core model, applying fixed latencies for every miss event that occurs and breaks the perfect instruction flow. Abstracting core performance, interval simulation drives the timing of each particular core without the need to keep a detailed track of all individual instructions. That way, Sniper reduces the simulation time by an order of magnitude compared to detailed cycle-accurate simulators [83], within an error of 5% on average for both single-threaded and multi-threaded applications.

Figure 3.1 represent a high-level organization. Functional model feeds the simulator with instruction streams using Pin as an instrumentation library. In case of multi-threaded OpenMP applications, there are as many instructions streams as running threads. Interval simulator models a set of cores by keeping an instruction 'window' per core as an approximation of ROB in out-of-order pipelines. Under the perfect conditions, without any miss events, instructions are dispatched at the maximal dispatch rate (*dispatch width*) taking into consideration instructions dependencies and execution latencies. Still, miss events coming from Branch and Memory Hierarchy Modules break this smooth instruction flow. Each of this particular miss events and latencies feed back into the analytical core model so it

can recalculate the timing for each interval. Core modules run concurrently and independently reducing the complexity of simulating multi-threaded applications.

Sniper integrates the McPAT [45] library for power and area modeling of CMPs. McPAT is based on CACTI [84], an integrated model for estimating power, area, and performance tradeoffs when designing memory subsystems. To estimate static power and area of core components, McPAT uses the configuration file from Sniper to provide necessary data, such as cache sizes, associativities, BP hardware budget, etc. At the end of the simulation, Sniper redirects its output data, such as the number of cache accesses, reads, misses, branch mispredictions, load and store instructions, so that McPAT can estimate dynamic power.

We use Sniper simulator in this thesis for several reasons. First, it is free to use and well established in academia research. Second, it models in detail all the core components we are interested in: instruction cache, branch predictor, and BTB. Next, it provides an infrastructure fast and accurate enough for us to simulate the entire applications, not just the representative regions. That way, we are able to evaluate both serial and parallel code sections. Finally, Sniper comes with tested and validated configuration files and tools making it attractive for our analysis of ACMP design.

3.2.2 TaskSim

Unfortunately, Sniper does not simulate accurately contention on shared resources. To get the better speedup at the cost of accuracy, it implements *lax synchronization* where communication on a shared resource is not performed on every access, but only on those where the receiver is behind the sender. Since our next step was to evaluate the idea of sharing the I-cache among multiple lean cores, we had to switch to another simulation framework that will accurately model the contention on shared interconnection networks. Still, as we surveyed a set of existing simulators, we did not find one that had a front-end pipeline modeled at such level of detail that allowed us to reason about the baseline and shared I-cache organizations. For example, having a pipelined front-end implementation is crucial in our analysis since an access to the shared I-cache can take multiple cycles. Also, the core front-end includes a set of line buffers that behave as a micro-cache or loop-buffer [85, 86] reducing the number of accesses to the I-cache (private or shared). Our core

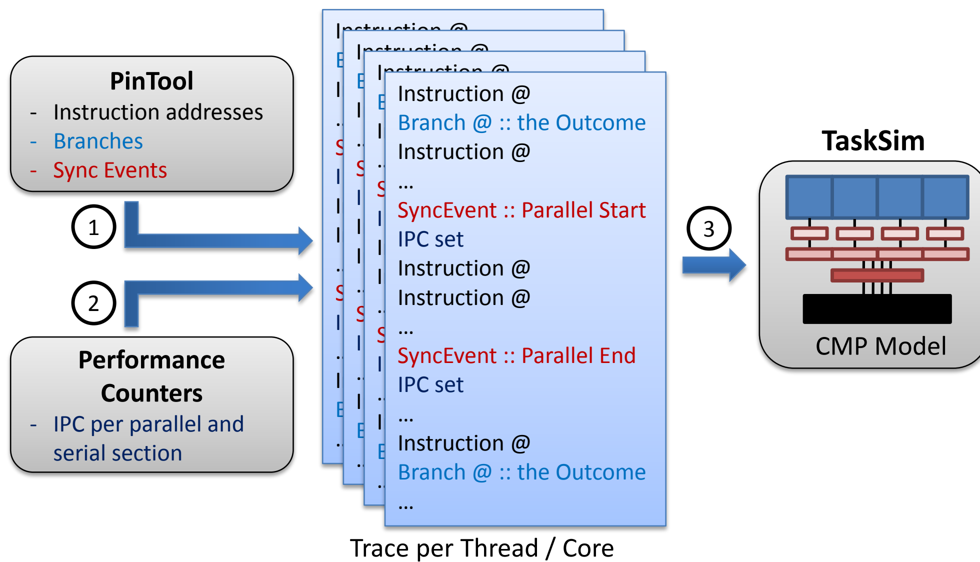


Figure 3.2: Temporal flow of a simulation process with TaskSim.

implementation thus had to model these features together with the rest of front-end structures in a cycle-accurate way.

For that reason, we turned to TaskSim [87], a trace-driven cycle-level simulator for parallel architectures running multithreaded applications. We use Pin [81] as the instrumentation framework for tracing the benchmarks. At runtime, our PinTool creates a trace file per thread storing the sequence of executed instruction addresses. For branch instructions, beside the address, it also stores its outcome (taken or not-taken) as well as branch target address. That way, we store all the information needed for reproducing the instruction stream. To resolve the weaknesses of trace-driven simulation such as inter-thread ordering and synchronization, we introduce synchronization events inside the trace files. We implement five events that cover all OpenMP primitives in the evaluated workloads: parallel start, parallel end, wait and signal on critical sections and semaphores, and barrier. The simulation framework thus has a double role. First, it models the entire CMP of interest, reads the trace files, and sends the requests to the I-caches for every fetch address. Second, it mimics the runtime system by managing the state of every thread according to the synchronization events in order to reproduce the same static scheduling of the application running in the real machine.

Figure 3.2 illustrates the simulation process. PinTool produces the traces, one per thread, capturing the instruction stream (step ①). Using performance counters from the real executions, we add IPC values to the traces, for each parallel and

sequential code section (step ②). Finally, TaskSim reads the traces and models the entire CMP (step ③).

Modeling a CMP organization inside TaskSim is based on defining a set of modules and their interconnections. Module can be any entity able to receive a request and send a response, such as cores, caches, crossbars, buses, DMAs, memories, etc. Any two modules can be interconnected through the input-output pair of ports so that the input of `Module0` is connected to the output of `Module1` and vice versa. Each input port has a queue to store the requests, defined latency after which the request on top can be processed, and defined width so the engine can calculate how many cycles is needed for a given request to be transfered through the port. For example, if the request size is 64 B and it has to be sent through the port which is 32 B wide, while respond latency of the module is 5 cycles, it will take in total 7 cycles for the current module to process the request. Knowing this, each module signals the simulation timing engine what is the earliest point in time to be awaken, avoiding time wasting process of awakening each module every cycle. This mechanism is crucial for our analysis of shared I-cache among cores in a CMP, giving us the ability to simulate the interconnection network between multiple cores and shared I-cache in a cycle-accurate way. With satisfying level of details, we evaluate different tradeoffs regarding interconnection topology, latency, and contention, port width, level of cache multibanking, etc. We provide more details on simulation setup in the following Chapters.

3.2.3 GPUSim

To evaluate multi-GPU systems, we use yet another simulation framework. It is an NVIDIA proprietary, cycle-level, trace-driven simulator for single and multi-GPU systems. The idea is similar to TaskSim, where modules communicate through the input and output ports, exchange the signals and packets, with the separate functional model that drives the logic runtime policies. To generate execution traces, we use SASSI instrumentation tool [88].

3.3 Benchmark Suites

3.3.1 Workloads for Shared-memory CMPs

We use 29 workloads from three benchmark suites to evaluate the benefits of tailoring the core front-end structures for HPC. Additionally, we analyze a set of 12 desktop applications to identify the differences between them and HPC workloads. An overview of benchmarks is given in Table 3.1.

- **ExMatEx applications.** This suite stands as a recent set of eight HPC applications with real scientific workloads, numerically intensive kernels and kernels bounded by memory, I/O, and communication [89]. We use default input parameters for each of these workloads. Half of the applications from this suite are implemented using OpenMP while the other half combine MPI and OpenMP.
- **SPEC OMP 2012 benchmark suite.** An objective and representative set of HPC workloads for measuring the performance of shared-memory CMPs [90]. The suite covers 11 scientific and engineering applications, from computational fluid dynamics to image manipulation, although, the suite has three more applications which are identical to the corresponding ones from the following NPB suite. These benchmarks were analyzed using the `reference` input set.
- **NAS Parallel Benchmark suite.** NPB suite is a set of 10 pseudo-applications derived from computational fluid dynamics apps [91]. Developed and maintained by NASA, it is a widely-used and standard set of HPC workloads. We use the `C` input set for analysing this suite.
- **SPEC CPU INT 2006 suite.** These benchmarks represent a standard set of applications for measuring the system's processor and memory performance [38]. This suite is included as a comparison between HPC and desktop applications. Using `reference` input set, we analyze only integer benchmarks (all 12 of them) since floating point workloads are derived from scientific applications and many of them are already covered by the SPEC OMP suite.

Suite	Benchmark	Programming Model	Input size
ExMatEx	ASPA	OpenMP	default
	CoEVP	OpenMP	default
	CoGL	OpenMP	default
	CoHMM	OpenMP	input2.txt file
	CoMD	MPI+OpenMP	default
	CoSP	MPI+OpenMP	1024 matrix
	LULESH	MPI+OpenMP	-s=200 -i=10
	VPFFT	MPI+OpenMP	default
SPEC OMP	md	OpenMP	reference
	bwaves	OpenMP	reference
	nab	OpenMP	reference
	botsalgn	OpenMP	reference
	botsspar	OpenMP	reference
	ilbdc	OpenMP	reference
	fma3d	OpenMP	reference
	swim	OpenMP	reference
	imagick	OpenMP	reference
	smithwa	OpenMP	reference
	kdtree	OpenMP	reference
NPB	BT	OpenMP	C
	CG	OpenMP	C
	DC	OpenMP	C
	EP	OpenMP	C
	FT	OpenMP	C
	IS	OpenMP	C
	LU	OpenMP	C
	MG	OpenMP	C
	SP	OpenMP	C
	UA	OpenMP	C
SPEC CPU INT	astar	sequential	reference
	bzip2	sequential	reference
	gcc	sequential	reference
	gobmk	sequential	reference
	h264	sequential	reference
	hmmer	sequential	reference
	libquantum	sequential	reference
	mcf	sequential	reference
	omnetpp	sequential	reference
	perlbench	sequential	reference
	sjeng	sequential	reference
	xalan	sequential	reference

Table 3.1: Evaluated shared-memory OpenMP benchmarks.

Benchmark	**Kernels**	**Time-weighted Average CTAs**	**Memory Footprint (MB)**
ML-GoogLeNet-cudnn-Lev2	1	6272	1205
ML-AlexNet-cudnn-Lev2	1	1250	832
ML-OverFeat-cudann-Lev3	1	1800	388
ML-AlexNet-cudnn-Lev4	1	1014	32
ML-AlexNet-ConvNet2	1	6075	97
Rodinia-Backprop	2	4096	160
Rodinia-Euler3D	346	1008	25
Rodinia-BFS	24	1954	38
Rodinia-Gaussian	510	2599	78
Rodinia-Hotspot	1	7396	64
Rodinia-Kmeans	3	3249	221
Rodnia-Pathfinder	20	4630	1570
Rodinia-Srad	4	16384	98
HPC-SNAP	118	200	744
HPC-Nekbone-Large	300	5583	294
HPC-MiniAMR	33	76033	2752
HPC-MiniContact-Mesh1	500	250	21
HPC-MiniContact-Mesh2	127	15423	257
HPC-Lulesh-Unstruct-Mesh1	2000	435	19
HPC-Lulesh-Unstruct-Mesh2	200	4940	208
HPC-AMG	88	241549	3744
HPC-RSBench	1	7813	19
HPC-MCB	1	5001	162
HPC-NAMD2.9	1	3888	88
HPC-RabbitCT	1	131072	524
HPC-Lulesh	105	12202	578
HPC-CoMD	350	3588	319
HPC-CoMD-Wa	350	13691	393
HPC-CoMD-Ta	350	5724	394
HPC-HPGMG-UVM	359	10436	1975
HPC-HPGMG	317	10506	1571
Lonestar-SP	11	75	8
Lonestar-MST-Graph	87	770	86
Lonestar-MST-Mesh	71	895	75
Lonestar-SSSP-Wln	1000	60	21
Lonestar-DMR	3	82	248
Lonestar-SSSP-Wlc	1300	163	21
Lonestar-SSSP	102	1046	38
Other-Stream-Triad	5	699051	3146
Other-Optix-Raytracing	1	3072	87
Other-Bitcoin-Crypto	1	60	5898

Table 3.2: Evaluated CUDA applications with memory footprint and time weighted average number of thread blocks available during execution.

3.3.2 CUDA workloads for GPU analysis

We study the scalability of multi-socket NUMA GPUs using 41 workloads taken from a range of production codes based on the HPC CORAL benchmarks [92], graph applications from Lonestar [93], HPC applications from Rodinia [94], in addition to several NVIDIA in-house CUDA benchmarks. This set of workloads covers a wide spectrum of GPU applications used in machine learning, fluid dynamic, image manipulation, graph traversal, and scientific computing. For some of the workloads we obtain traces with different inputs (like `HPC-MiniContact-Mesh1` and `HPC-MiniContact-Mesh2`) or different algorithm being implemented to solve the problem (like `HPC-CoMD-Wa` and `HPC-CoMD-Ta`). Each of our benchmarks is hand selected to identify the region of interest deemed representative for each workload, which may be as small as a single kernel containing a tight inner loop or several thousand kernel invocations. We run each benchmark to completion for the determine region of interest. Table 3.2 provides both the memory footprint of these workloads as well as the average number of active CTAs in the workload (weighted by the time spent on each kernel) to provide a representation of how many parallel thread blocks (CTAs) are generally available during workload execution.

Chapter 4

HPC Workload Characterization

This chapter presents a characterization of HPC applications running on a shared-memory CMPs. Our focus is on the core front-end structures such as instruction cache (I-cache), branch predictor (BP) and branch target buffer (BTB). We evaluate three HPC benchmark suites and compare them with traditional desktop applications, found in SPEC CPU INT. Moreover, this work analyses the difference between serial and parallel code sections inside HPC workloads. We find that HPC workloads have fewer branch instructions, more biased and backward taken conditional branches, smaller instruction footprints, and longer basic blocks. Those findings suggest the use of smaller I-caches with wider lines, smaller BPs with loop BPs, and smaller BTBs. Still, observing that serial parts in HPC benchmarks are more close to desktop applications, we advocate for an ACMP design from the core front-end perspective. Downsizing the front-end structures on throughput-oriented cores our estimations show 16% of area and 7% of power savings with no impact on performance.

4.1 Microarchitecture Independent Characterization

Here, we provide intrinsic code characteristics of HPC applications. We focus on aspects affecting the core front-end: branches, instruction footprints, and basic

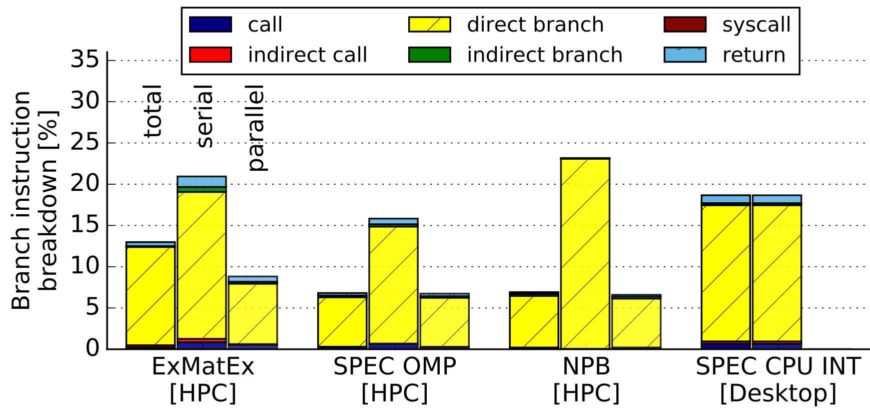


Figure 4.1: Dynamic branch instruction breakdown for each benchmark suite as the percentage of total instructions.

blocks. We also point to a difference between sequential and parallel code sections we observed among workloads.

4.1.1 Branch Instructions

We start our analysis determining the dynamic mix of branch instructions. The PinTool inspects every branch instruction and counts its frequency and type.

Figure 4.1 shows the dynamic branch instruction breakdown. All branch instructions factor out 13% of the total dynamic instruction mix for ExMatEx suite and around 7% for SPEC OMP and NPB, compared to 19% on average for SPEC CPU INT workloads (*total* bars). This indicates that HPC workloads probe branch predictors less often. The number of system calls is negligible. Indirect jumps (both branches and calls) are rare except for `EP`, `UA`, `md`, `kdtree`, and `CoEVP`. On average, they are less than 0.5% of all branches, and up to 2.5% in case of `CoEVP`. The majority of dynamic branch instructions are conditional and unconditional direct branches. This figure points a big difference between serial and parallel code sections inside the HPC applications. Closer to SPEC CPU INT, sequential parts have almost 3× more branch instructions than parallel parts on average.

Figure 4.2 presents a more detailed analysis of dynamic conditional direct branches. It gives a stacked bar for each suite showing the distribution of branches depending on the percentage of times they are taken. HPC workloads have between 90% (in case of NPB) to 80% (in case of ExMatEx) of dynamic branches dominantly either

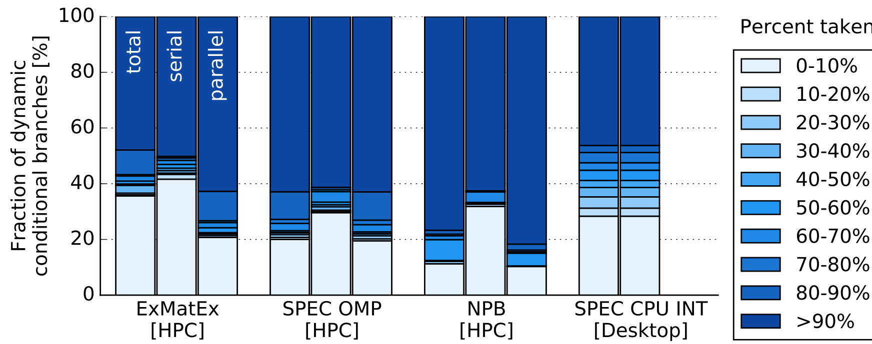


Figure 4.2: Distribution of branch directions. Conditional branches are dominantly decided in one direction, either taken or not taken. Desktop applications have more evenly distributed directions of conditional branches.

Table 4.1: The average percentage of backward and forward taken branches per benchmark suite.

Suite	Serial code		Parallel code	
	backward	forward	backward	forward
ExMatEx	72%	28%	69%	31%
SPEC OMP	73%	27%	74%	26%
NPB	71%	29%	80%	20%
SPEC CPU INT	56%	44%		

taken or not taken. On the other side, SPEC CPU INT applications have more distributed directions of conditional branches. Interestingly, serial and parallel code sections have similar behavior, except that not-taken branch instructions are more frequent in sequential code. Additionally, Table 4.1 breaks down all taken branches on backward and forward ones. While for HPC suites around 75% of taken branch instructions jump backwards, SPEC CPU INT benchmarks have almost the same number of forward and backward taken branches. Again, serial and parallel code sections show similar '75-to-25' ratio between backward and forward taken branches. These results show bias in direction with the potentially high predictability of branches among HPC workloads. They suggest that the use of a simple and smaller branch predictor would yield a low misprediction ratio, especially in parallel code regions. We analyze this assumption in Section 4.2.1.

4.1.2 Instruction Footprint

To find out the required I-cache design, we analyze the sizes of both static and dynamic instruction footprints. We use a PinTool that stores the size of each basic

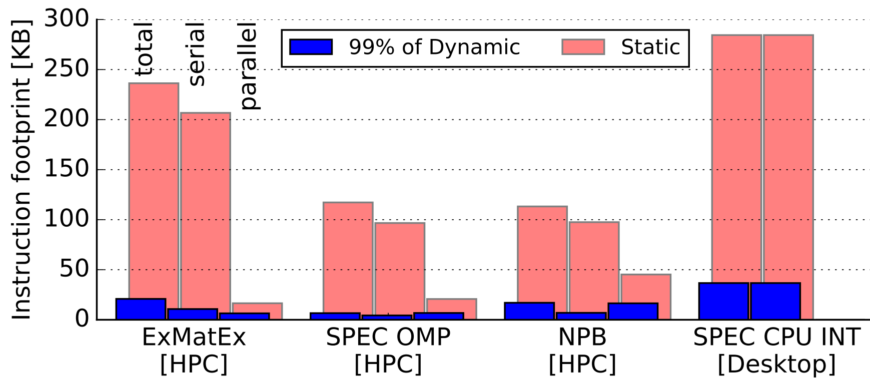


Figure 4.3: Static instruction footprint and memory we need to store 99% of dynamic instructions per benchmark suite.

block in bytes and counts how many times that basic block has been executed. That way, we find the static and dynamic instruction footprints per basic block and, thus, for the whole application.

Figure 4.3 shows the total static instructions footprint and the amount of memory needed to fit 99% of dynamic instructions, averaged per benchmark suite. Workloads from SPEC OMP and NPB suites have small static code size, up to 252 KB (for `UA`) and around 121 KB on average. Workloads from ExMatEx suite have larger static instruction footprint, up to 800 KB for `VPFFT` and 242 KB on average. These benchmarks are more recent, close representatives of real applications, linked with external libraries (such as LAPACK, BLAS, FFTW) that increase total instruction footprint. Among HPC workloads, sequential code has larger static instruction footprint, but still smaller than desktop applications. There is high spatial code locality. Most of the HPC workloads (17 out of 29) fetch almost 100% of instructions from one or two KB of memory. Still, cases such as `LULESH` and `CoGL` from ExMatEx or `BT` from NPB suite, fetch between 60% and 90% of instructions from 16 KB of memory. On average, HPC workloads in parallel execute 99% of instructions from just 14 KB of memory. Serial code sections also show high spatial locality, even higher compared to parallel sections for SPEC OMP and NPB suites. Among these benchmarks, the total number of instructions executed sequentially is small, thus the existence of any loop(s) (the majority of taken branches is backward-taken according to Table 4.1) results in high code spatial locality. In this case, the serial code inside parallel HPC applications show different behavior from SPEC CPU INT.

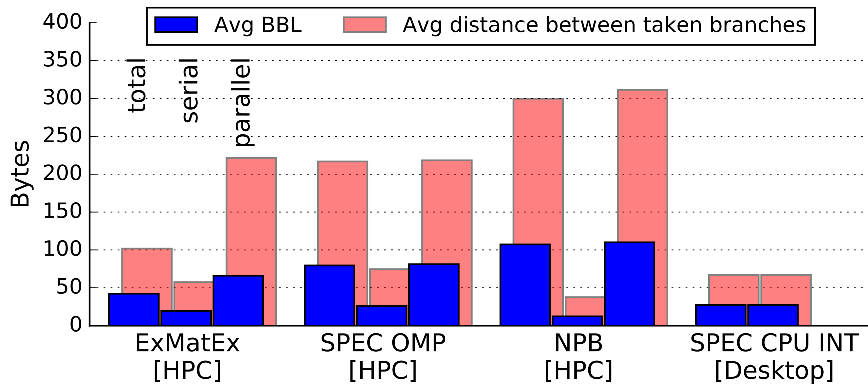


Figure 4.4: Average dynamic basic block length and distance between taken branches for each benchmark suite.

These results indicate that most of the HPC benchmark's dynamic code fit in less than 32 KB of memory. For many of them, even 4 KB of memory is enough to store almost every instruction. HPC applications spend most of their time inside loops, so few basic blocks are fetched and executed over and over again. Nevertheless, to know exactly how these characteristics impact the number of I-cache misses, we need to observe temporal behavior as well. We cover this analysis later, in Section 4.2.3.

4.1.3 Basic Blocks

Due to the low frequency of branch instructions, we expect to find long basic blocks in HPC benchmarks. Traditional desktop and server applications are known to have short basic blocks [95, 96]. Our analysis confirms this. Many complex features are implemented in today's CPUs to overcome the problem of short basic blocks and increase fetch bandwidth, such as multiple branch predictions per cycle, instruction alignment, and a trace cache. Tailoring an HPC core, these may not be needed.

Figure 4.4 shows the average dynamic basic block length and the average distance between taken branches per benchmark suite. The average basic block size for HPC applications is around 78 bytes. Some of them have very long basic blocks, for example, `BT` (312 B), `swim` (152 B), and `LULESH` (126 B). We present per-benchmark values later in Section 5.1. The distance between taken branches is even longer. It suggests the usage of wider I-cache lines that would still have high

usefulness and keep fragmentation low. For those benchmarks where basic blocks are not long, reuse distance is short. That is the case with `CoHMM`, `CoSP`, `botsspar`, `CG`, and `IS`, where the average basic block size is around 32 B, but the majority of them are executed with a reuse distance between one and two basic blocks. Once fetched, a wide cache line would be frequently reused without sending new fetch requests to the I-cache, acting as a prefetch buffer [97]. Compared with SPEC CPU INT applications, HPC workloads have around 4× longer basic blocks and 5× longer distance between taken branches with parallel code sections. Sequential parts are similar to desktop applications. These results are important for a design of the I-cache, as we show in Section 4.2.3.

4.1.4 Difference Between Sequential and Parallel Code Sections in HPC Workloads

Our previous measurements demonstrate that ExMatEx benchmarks have slightly bigger code sizes, less biased branches, and shorter basic blocks compared to SPEC OMP and NPB. There are two reasons for such a behaviour.

First, this suite includes benchmarks with a considerable amount of instructions executed in sequential regions bringing its characteristics overall closer to SPEC CPU INT. Run on an eight-core CMP, `CoEVP` has a master thread that executes around 35% of its instructions sequentially, between parallel sections. The similar behaviour is observed for `CoMD` (8% of instructions is executed in sequential parts), `CoSP` (9%), and `LULESH` (11%) applications, all from the ExMatEx suite. Comparing basic block lengths, `CoMD` and `LULESH` have 2× and 3× longer basic blocks in parallel than in sequential code sections, respectively. Among SPEC OMP and NPB workloads, master thread executes less than 1% of all instructions in sequential regions, except for `nab` and `fma3d` (around 4% in sequential parts). On the previous graphs, the *total* bars are always between *serial* and *parallel* for ExMatEx, while for SPEC OMP and NPB *total* is almost the same as *parallel*.

Second, ExMatEx benchmarks include many external libraries which increase their instruction footprint. This consequently increases the number of branch instructions, and as we shall see, complexity in predicting them, increasing the number of misses in the appropriate front-end structures. If our analysis were done only with

SPEC OMP and NPB benchmarks, we could have ignored the impact of these facts, leading us to some wrong conclusions and findings.

On the other side, the amount of instructions executed in serial directly depends on the number of threads running the application and the size of input set. For example, running `fma3d` and `nab` benchmarks from SPEC OMP with `train` inputs and eight threads gives around 25% of all instructions executed in serial by master thread. With the same inputs as we use here (`reference`) but running 64 threads, the fraction of serial code increases to 18% and 19% (from 4% with eight threads). Today, Intel's Xeon Phi cards and IBM's Power8 CMPs support around 200 threads. As the number of cores and/or hardware threads per CMP increases, handling the serial parts of parallel applications may become crucial. Our analysis so far shows not just that the HPC benchmarks are different from desktop, but that also serial code sections are different from parallel inside an HPC application.

4.2 Microarchitecture Dependent Characterization

Driven by the observations in the previous section, we analyze how to accommodate the core front-end structures for HPC applications. We focus on branch predictors, branch target buffers, and instruction caches.

4.2.1 Branch Predictor

Previous findings demonstrate the existence of a small number of biased branches in HPC applications. It suggests the high predictability and use of simpler and smaller branch predictors that can provide the same performance as the ones we can currently find in today's CPUs. To evaluate this idea, we implement a PinTool with three different branch predictors:

- **gshare** - branch prediction that uses one global history table and branch history register (BHR) XORed with branch address to index the history table [98].

Table 4.2: Size parameters and hardware cost of evaluated branch predictors. Parameter n stands for the number of address bits used to index the tables, and parameter m stands for branch history length.

Predictor	**Hardware cost (bits)**	**Size parameters**	
		~2 KB (small)	~16 KB (big)
gshare	2^{m+1}	$m = 13$	$m = 16$
tournament	$2^n(m+2) + 2^{m+2}$	$n = 10, m = 8$	$n = 12, m = 14$
TAGE	according to [101]	2 tables	12 tables

- **tournament** - the branch predictor implemented in the Alpha 21264 processor [99]. It has two branch predictors, one based on private and the other on global history tables, and the one which is currently more successful predicts the outcome of a branch.
- **TAGE** - a modern branch predictor that relies on tagging the entries and capturing different global history lengths [100]. It uses a base predictor (bimodal) and a set of tagged tables indexed using different history lengths that form a geometric series.

To make a fair comparison between branch predictors, we evaluate configurations that have the same hardware cost. Table 4.2 gives an overview of the parameters used for different branch predictors, so they have approximately the same size. We define two sets of configuration parameters, *small* with a 2 KB hardware budget and *big* with 16 KB. We take this as a reasonable assumption in a lean-core design given that on the 2nd Championship Branch Prediction competition [101], 4 KB and 32 KB budgets are used for heavyweight cores.

Since HPC workloads spend most of their time inside loops and the majority of taken branches are backwards, we also check how a loop branch predictor (LBP) affects mispredictions when it is added to the small predictors analyzed here. An LBP tries to identify loops with a constant number of iterations. The prediction is by default provided by the base predictor, but, in cases where high confidence is achieved by the LBP, the prediction from an LBP is taken as the final decision. We implement a 64-entry LBP with an approximate hardware budget of 512 B.

Figure 4.5 shows the number of branch mispredictions per kilo instructions (branch MPKI) for every branch predictor and three configurations per BP: big, small, and small with an LBP (bars with prefix L on a graph). There are several things to

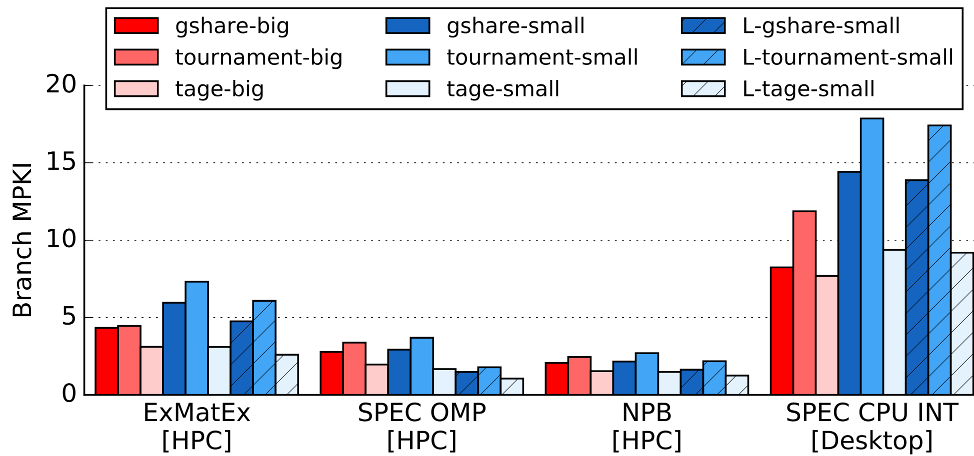


Figure 4.5: Branch MPKI for different branch predictor configurations and benchmark suites.

observe here. First, Figure 4.5 demonstrates the difference between desktop and HPC workloads for the same configurations and types of branch predictors. As pointed out before, HPC workloads have fewer branch instructions per execution and more biased branches. This results in SPEC CPU INT applications having around 3× higher branch MPKI compared to ExMatEx ones, and around 5× compared to SPEC OMP and NPB ones. For every HPC benchmark suite, sequential parts have higher branch MPKI than its parallel parts, but lower than SPEC CPU INT.

Second, it is clear that TAGE outperforms the other two branch predictors for all cases. This holds not just on per suite, but also, on per benchmark level. TAGE is much better in reducing the interference or aliasing which occurs when different branch instructions point to the same prediction bits. With the simple usage of lower address bits or XORing them with a history register, different branch instructions can map to the same prediction entry which reduces the effective usage of a prediction table. It can even be destructive if the branch instructions take different directions. By (partially) tagging its entries, TAGE eliminates (reduces) this effect. Also, TAGE has multiple components each for a different global history length from very short to very long. Compared to the other two branch predictors and for HPC benchmarks, TAGE provides almost the same branch MPKI values regardless of its size. With just a 2 KB hardware budget, small TAGE is better than 16 KB big gshare or tournament predictor.

Figure 4.5 also demonstrates how desktop and HPC applications are different in

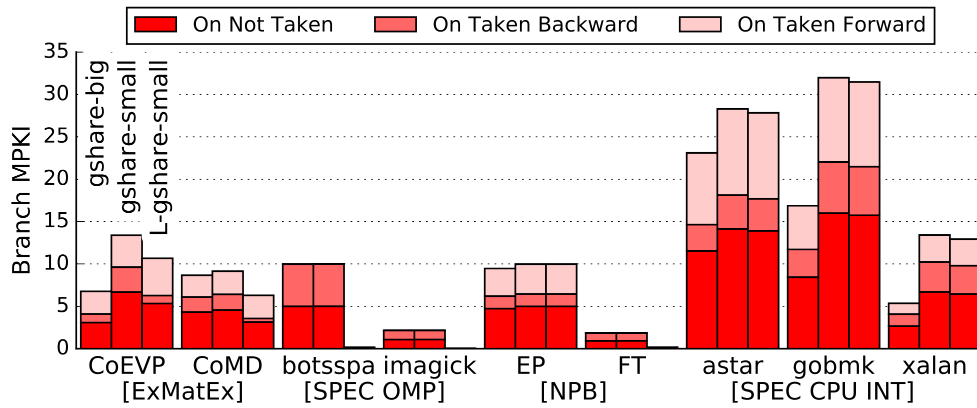


Figure 4.6: Branch MPKI breakdown for gshare branch predictor and a subset of workloads. We distinguish mispredictions on not taken, taken backward, and taken forward branches.

exploiting the LPB. For each benchmark suite, it shows the reduction of branch MPKI values when an LPB is implemented together with a small base predictor. It barely reduces the number of misses for desktop applications. On the other hand, HPC applications, both sequential and parallel code sections, benefit from an LPB. Still, reducing the size of branch predictor increases the MPKI values in serial parts. These results show that a core used to execute parallel HPC code should have a branch predictor tuned differently than the one used to run desktop applications. With biased and mostly backward taken branches, long basic blocks, and low fraction of branch instructions in the instruction mix, smaller and properly configured branch predictors can be used in HPC cores without performance loss.

Figure 4.6 breaks down the branch MPKI values obtained with gshare predictor for a subset of workloads. A branch misprediction can be caused by a not taken, a taken backward, or a taken forward branch instruction. As expected, the presence of a loop BP reduces the number of branch mispredictions on taken backward branches, especially for HPC workloads. While it has a moderate effect on branch MPKI values for benchmarks like `CoEVP` and `CoMD`, in cases of `botsspar` and `imagick`, an LBP eliminates the branch mispredictions, not just the taken backward, but also the not taken ones. After the last loop iteration when the branch should not be taken, a two-bit entry in a gshare table would miss because the saturating counter is in a *strongly-taken* prediction state, while an LBP knows exactly how many iteration that loop will execute. Still, there are some HPC benchmarks where the presence of a loop BP has no effect on a branch MPKI value such as in case of `EP`. Looking at the SPEC CPU INT benchmarks, taken

backward misses exist but they are not reduced by an LBP since those are not part of loop structures. It is interesting to note that the majority of all mispredictions comes from the not taken branches, for all benchmark suites.

TAGE branch predictor shows similar behavior as gshare on Figure 4.6 but with lower branch MPKI values and without the difference between *big* and *small* configurations for HPC benchmarks. An LBP is also beneficial for TAGE but mostly reducing mispredictions on not taken branches. When the control flow inside the loop is regular, TAGE predictor is able to predict loops with constant number of iterations, just as an LBP. On the other hand, when the control flow in the loop body is changeable, TAGE predictor may fail to correctly predict the exit of the loop [101].

4.2.2 Branch Target Buffer

The branch predictor provides information about whether the next branch will be taken or not taken. Still, it does not supply the target address in case the branch is predicted as taken. For that, we use the branch target buffer (BTB) which is implemented as a cache that stores the branch target address for taken branches. We use the current instruction address to index and lookup in the BTB and, if the address is found, then the instruction being fetched is a taken branch, and the data part of the entry contains the next PC after the branch. In the BTB, we store only branches predicted as taken since not-taken ones will continue fetching from the next sequential instruction. With a correctly predicted outcome of a branch and a correct target address stored in the BTB, we have a zero branch penalty.

Figure 4.7 shows how the number of misses in the BTB depends on its size and associativity. Changing the BTB size from 256 to 1024 entries, has no impact on the number of misses for HPC applications. High associativity is needed to reduce the aliasing problem, especially for ExMatEx benchmarks, mostly due to a simple modulo indexing to the BTB. Our previous findings state that HPC benchmarks have a small number of branches and they are strongly biased. Once when BTB stores a destination address for a taken branch, that branch is probably going to be taken the next time it occurs with the same destination address. This results in the same MPKI values regarding the size of the BTB. Desktop applications show higher BTB MPKI values for the same BTB configurations compared to HPC

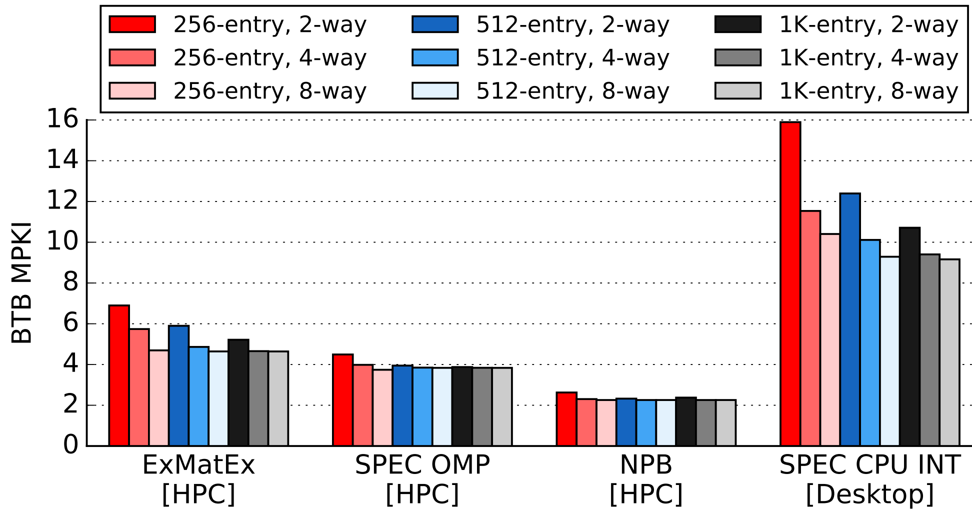


Figure 4.7: BTB MPKI for different number of entries and associativity. We use branch address to index BTB.

applications. Bigger branch target buffers provide significantly lower BTB MPKI, such as in cases of `gcc`, `gobmk`, and `sjeng`.

4.2.3 Instruction Cache

Section 4.1.2 explains how HPC applications have a small dynamic instruction footprint. Moreover, most of the execution time is spent in loops where only a few basic blocks are executed many times. Figure 4.3 shows that for parallel parts of HPC benchmarks about 99% of all instructions are fetched from less than 32 KB of memory. On the other side, an I-cache factors out a considerable part of power and area on lean cores. We check how different I-cache sizes and associativities impact the number of misses.

Our pintool simulates the I-cache behavior throughout the execution. In the beginning, it creates a cache structure with the specified characteristics such as cache size, line width, and associativity. We implement LRU replacement policy.

Figure 4.4 points that HPC applications have long basic blocks. Once we fetch an I-cache line, we extract the instructions sequentially, without accessing the I-cache again, until we reach the end of a line or a taken branch. Due to the long distance between taken branches and without any alignment techniques, even 128B-wide I-cache lines have a high usefulness, 71% on average. We define usefulness as the

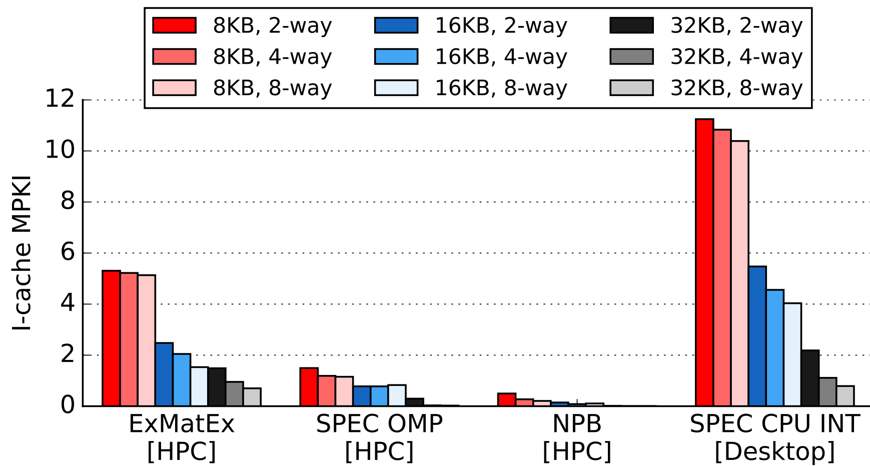


Figure 4.8: The average I-cache MPKI values for all benchmark suites. The cache line is 64 B.

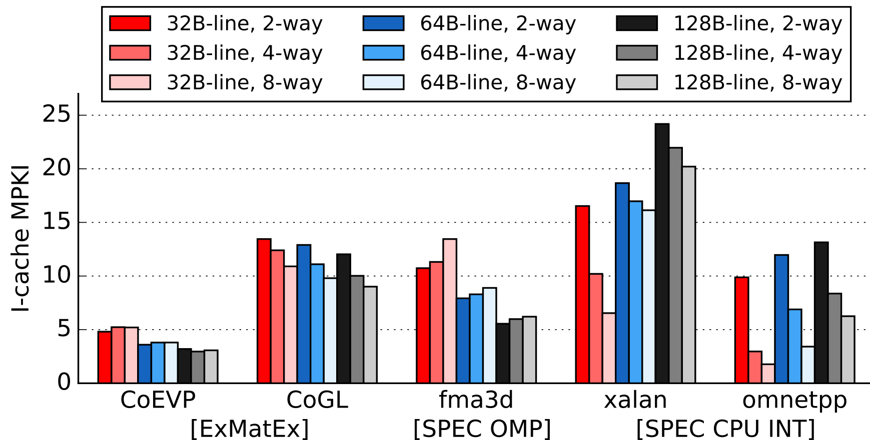


Figure 4.9: I-cache MPKI values for some specific benchmarks. The cache size is 16 KB.

number of different bytes accessed in a line out of the total line size. For the same line width, SPEC CPU INT has only 33% usefulness. Besides that, wider cache lines also reduce the number of accesses to the I-cache.

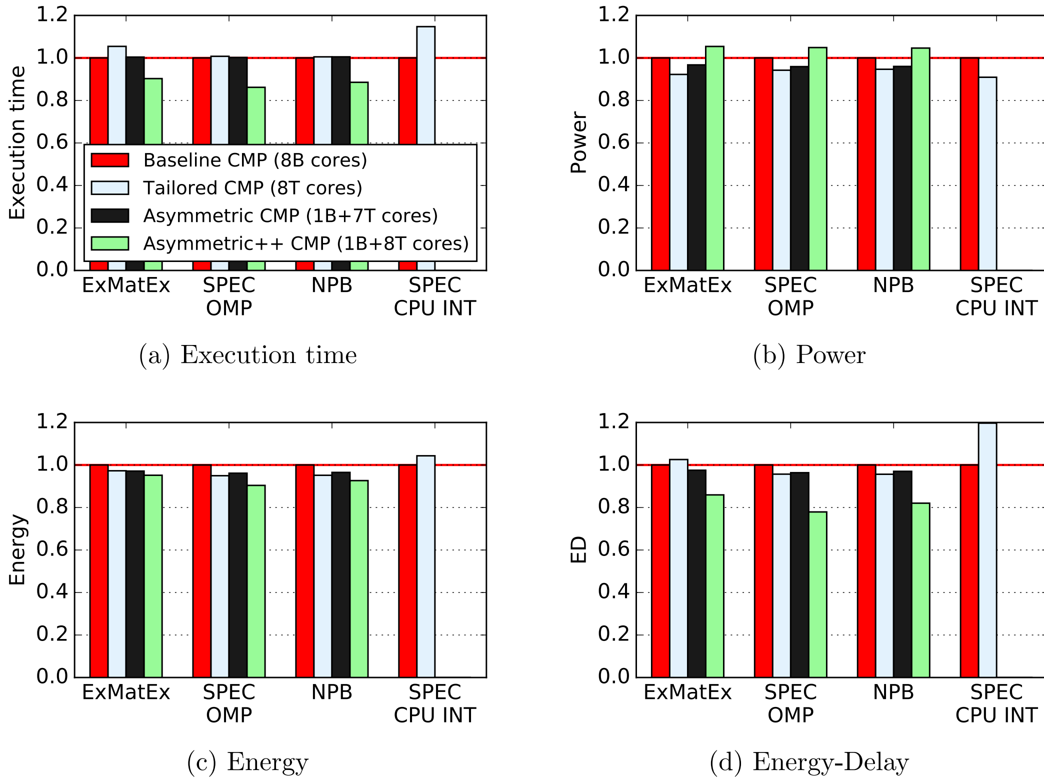
Figure 4.8 shows how the number of misses in the I-cache depends on its size and associativity, averaged per benchmark suite. Desktop applications, with their large static code footprints, need larger I-caches. Reduction in size is not an option for them since the use of a 16 KB I-cache increases the MPKI 2.5× compared to a 32 KB I-cache. For all benchmarks in SPEC OMP (except the `fma3d`) and NPB suites, even an 8 KB I-cache provides MPKI values bellow 1. ExMatEx applications stress more the I-cache. They have larger static and dynamic instruction

footprints, as we have seen in Figure 4.3. For them, an 8 KB I-cache is not an option. With high associativity and 128 B lines, a 16 KB I-cache increases the MPKI from 1 to 2 on average, compared with a standard-size 32 KB I-cache (for per-benchmarks values see Figure 5.2). Instructions executed in sequential regions miss by 50% more on average, and in case of `CoSP` up to 2×, compared to instructions from parallel regions.

There is an interesting observation analyzing the impact of the I-cache line width on the MPKI values. Figure 4.9 shows these dependencies for a subset of benchmarks. While wider lines reduce the number of misses in the I-cache for HPC applications, for SPEC CPU INT workloads is the opposite. For a fixed size and associativity, HPC applications miss by 16% less in a 128B-line than in a 32B-line I-cache. For the same comparison, SPEC CPU INT benchmarks have 19% more I-cache misses on average. With short basic blocks and short distance between taken branches, wider cache lines used with desktop applications have low usefulness and reduce the number of total cache lines available in a fixed-size I-cache. On the other side, HPC workloads benefit from wider I-cache lines, not just due to reducing the number of accesses to the I-cache but also from the high usefulness of wide lines. The existence of hot code regions, such as loops, forces the running thread to execute a few basic blocks multiple times. No matter how large static code is (due to external library linking or any other reason), dynamic instruction footprint remains small and able to fit in less than 32 KB of cache memory.

4.3 Impact on Performance, Power and Area

As our Section 4.1 highlights, HPC workloads have specific code characteristics. They have a low number of biased, and mostly backward taken branches. Dynamic instruction footprints are small and basic blocks long. Those results suggest a redimensioning of the core front-end structures for HPC, such as I-cache, branch predictor, and BTB. We use Sniper [82] to simulate the performance impact and McPAT [45] library to project power and area savings by tailoring the core front-end for HPC applications.



(a) Execution time (b) Power

(c) Energy (d) Energy-Delay

Figure 4.10: Normalized execution time, power, energy, and energy-delay (ED) product for different CMP configurations and averaged per benchmark suite. We analyse only cores and L2 caches since the rest of CMP is shared and same for all configurations. Asymmetric++ CMP has the same area budget as Baseline CMP.

4.3.1 Experimental Setup

We have selected the ARM Cortex-A9 configuration file from the McPAT bundle because it has been validated against real silicon and is representative of lean cores. It also has similar area footprint compared to an IBM BlueGene/Q core and recent research works consider ARM a potential player in the HPC market [33]. In Sniper, we configure an eight-core CMP with Cortex-A9-like cores, private 256 KB L2, and 4 MB shared L3 cache. HPC applications are run with eight threads while SPEC CPU INT are run sequentially. For our baseline core model, we use 32 KB, 64B-line I-cache, 16 KB tournament BP, and 2K-entry BTB. Based on the results from our previous Sections, for the alternative HPC-core design we simulate 16 KB, 128B-line I-cache, 2 KB tournament predictor with loop BP, and 256-entry BTB. We refer to it as a tailored core model.

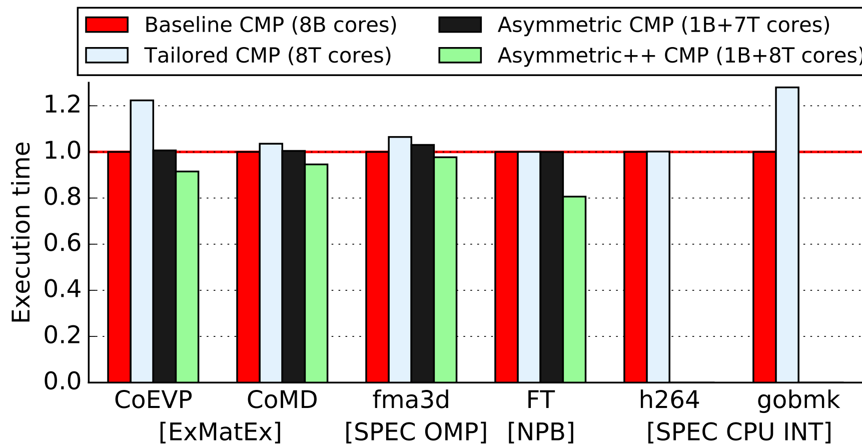


Figure 4.11: Execution time for a subset of benchmarks, normalized to a baseline CMP configuration.

4.3.2 Results

Figure 4.10a shows the average execution time normalized to the baseline CMP configuration. Figure 4.11 is similar, presenting the same metric for a subset of benchmarks. As expected, reducing the sizes of front-end structures is not acceptable for desktop applications, although, in some particular cases, it provides no performance degradation (like for `h264`). They need large branch prediction structures to handle complex branch instructions and large I-cache to store the code. SPEC OMP and NPB benchmarks increase their execution time by less than 1% when they are executed on a CMP made out of tailored cores compared to the baseline ones. Among NPB workloads, there is no a single benchmark with more than 3% of execution time increase, while for SPEC OMP, only `fma3d` demonstrates a significant performance loss of 6%, mostly due to the I-cache misses. Running ExMatEx benchmarks on an eight-core tailored CMP increases the execution time by almost 6% on average, hurting four out of eight workloads (`CoHMM`, `CoEVP`, `CoMD`, and `CoGL`). The highest is `CoEVP`, whose execution time goes up by 22%.

As we mentioned before, `CoEVP` benchmark with default inputs and running eight threads spends around 35% of its time inside the serial code. Binding the master thread to run on a baseline core, while the rest of threads run on tailored cores (Asymmetric CMP) provides the same performance as running this benchmark on an eight-core Baseline CMP. This shows that heterogeneity which already exists in HPC systems with accelerators, should be implemented even deeper, on a CMP

Table 4.3: I-cache, BP, and BTB share in total area and power budget on a Cortex-A9 core level. BP has 12-cycle miss penalty. The numbers are obtained using McPAT and CACTI tools with processing technology of 40 nm.

		Area[mm^2]	Power[W]
Baseline	**Total core**	**2.49 (100%)**	**0.85 (100%)**
	I-cache (32 KB, 64 B line)	0.31	0.075
	Big BP (16 KB)	0.14	0.032
	BTB (2K entries)	0.125	0.017
Tailored	**Total core**	**2.11 (84%)**	**0.79 (93%)**
	I-cache (16 KB, 128 B line)	0.14	0.049
	Small BP with LBP (2.5 KB)	0.04	0.011
	BTB (256 entries)	0.022	0.002

level. Used as a stand-alone component, accelerators as Intel Xeon Phi may suffer from either under-dimensioning the master core (and slowing down the serial part), or over-dimensioning the workers (and wasting resources on them). Asymmetric CMP designs are already present in different markets (IBM's Cell or ARM's big.LITTLE), and our results show that a similar approach has an advantage in HPC as well.

In the baseline configuration, a 32 KB I-cache with 64 B lines contributes with 12% and 9% of the total area and power core budget, respectively. A 16 KB branch predictor, implemented as a tournament predictor in McPAT and thus in Sniper for consistency, factors out around 5% in area and 4% of core power. A 2K-entry BTB in baseline contributes with 5% and 2% of the total core area and power budget. Table 4.3 provides these numbers in absolute values.

As results suggest the use of a smaller I-cache, smaller BP with LBP, and a BTB with less entries, we compare the baseline numbers with the ones obtained configuring a 128B-line 16 KB I-cache, 2 KB BP, and a 256-entry BTB. Reducing the sizes of these hardware structures gives 16% savings in area and 7% savings in power at the core level.

Saving this amount of area per core on an asymmetric eight-core CMP, opens the opportunity to add an extra core. With the abundant TLP, we can scale performance with core count under the same area budget. Figures 4.10a and 4.11 compare the execution times of HPC workloads on an Asymmetric++ CMP composed out of one baseline and eight tailored cores to a Baseline CMP composed out

of eight baseline cores. For the same area budget, Asymmetric++ CMP provides 12% time reduction on average and up to 20% for `FT`.

The rest of plots on Figure 4.10 show normalized power, energy, and energy-delay (ED) product for different CMP configurations. There is an interesting tradeoff present on Figure 4.10b. Power is estimated as a sum of the static and dynamic power of private structures: cores and L2 caches. Downsizing the front-end structures we save static power and reduction of dynamic power comes mostly due to the increased execution time for a Tailored CMP. With an additional core, Asymmetric++ CMP increases the power budget by 4% compared to Baseline CMP, on average. With 12% performance improvement and within the same area, this translates into 8% of energy savings and reduction of ED product by 18%.

Chapter 5

Sharing the I-cache among Lean Cores

In the previous chapter, we have shown the difference between serial and parallel code sections inside HPC applications. Parallel code regions have less branch instructions, longer basic blocks, and smaller instruction footprints. Here, we evaluate the idea of having an I-cache shared among lean cores that run parallel code. We motivate this idea by finding that dynamic instruction footprint for all running threads is 99% the same. Thus, sharing the I-cache we can potentially reduce the number of misses due to constructive code prefetching between the threads but also reduce the total area occupied by a set of private I-caches. On the other side, prolonged access time to the shared resource may hurt the performance. In this chapter we analyze this treadeoff focusing on the ACMP design used in HPC.

5.1 Sequential and Parallel Code within HPC applications

On an ACMP, the large core executes sequential code and it joins the workers executing parallel code regions. Using Pin [81] as an instrumentation library, we instrument only the master thread and characterize the HPC applications separating the serial and parallel sections by looking at the average basic block size

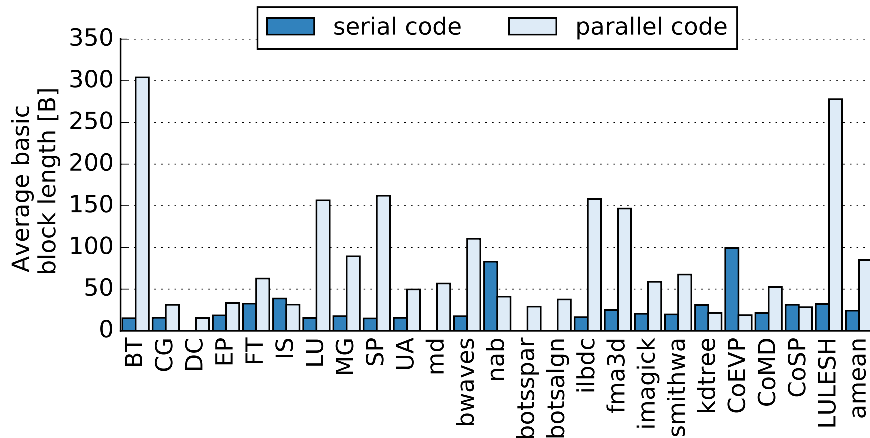


Figure 5.1: The average dynamic basic block length in serial and parallel parts of the code.

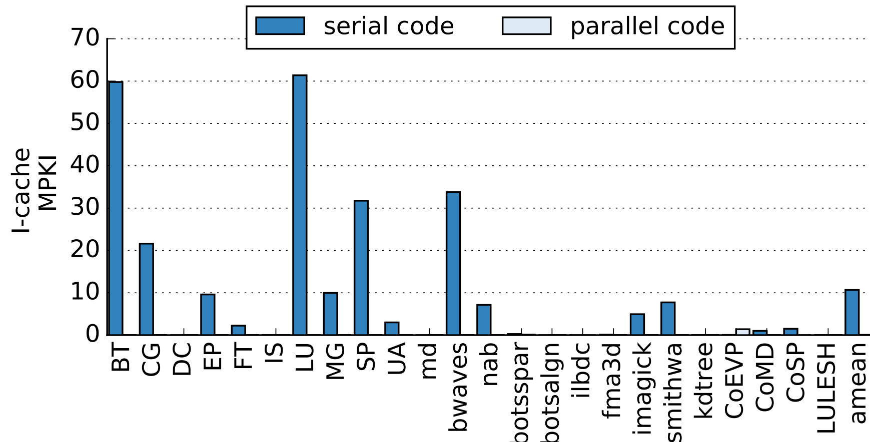


Figure 5.2: The I-cache MPKI values in serial and parallel parts of the code using a 32 KB, 8-way associative I-cache with 64 B lines, and LRU replacement policy. The I-cache MPKI values in parallel code are very low.

and the I-cache MPKI values. Here, we show the values for each evaluated workload, while sections 4.1.3 and 4.2.3 show the same numbers but averaged across benchmark suites.

Figure 5.1 shows the average dynamic basic block size for each workload we use in this evaluation (a subset of benchmarks described in Section 3.3.1). HPC applications have 3× longer basic blocks in parallel than in sequential code. This means that HPC benchmarks, while executed in parallel, provide high usefulness of the I-cache lines, increasing the useful fetch bandwidth without any techniques such as trace cache or multiple branch prediction per cycle [8, 11]. A single I-cache line fetched inside the parallel region contains more instructions to feed the core

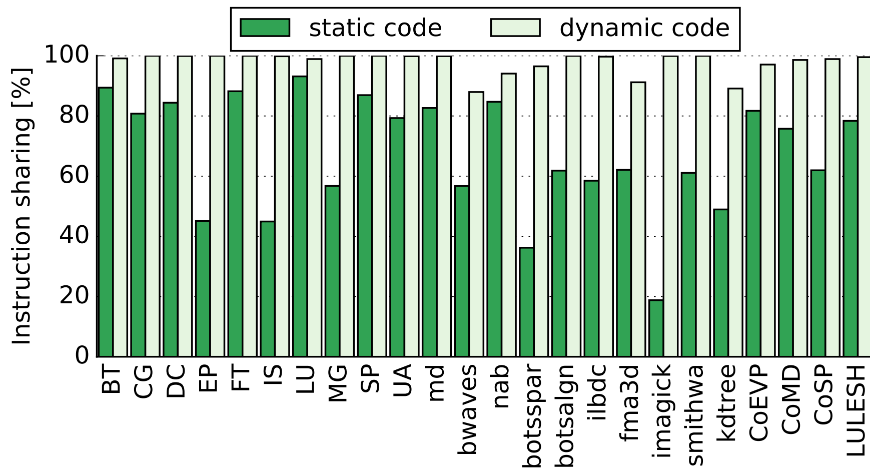


Figure 5.3: Percentage of instruction sharing across all threads running on an eight-core CMP per HPC benchmark (parallel sections only).

back-end than an I-cache line from a serial region. Still, there are benchmarks, such as `nab` and `CoEVP`, where basic blocks are longer in serial sections. We will refer to these interesting cases later in Section 5.5.5.

Figure 5.2 gives the I-cache MPKI values for each benchmark obtained in serial and parallel code regions. Not just that sequential code sections miss more in a standard-size 32 KB I-cache, but parallel code sections have I-cache MPKI values far below 1 (except for `CoEVP`). HPC applications spend most of their time inside loops, so few basic blocks are fetched over and over again, resulting in a few I-cache misses.

These findings point out the difference between sequential code executed by the large, master core and parallel code executed by all cores. With its aggressive back-end and short basic blocks, the large core needs quick access to the instruction memory to deliver enough instructions every cycle. On the other hand, lean cores have less demanding back-ends and 3× longer basic blocks, so a prolonged I-cache access latency is less likely to introduce additional stall cycles. Moreover, parallel code sections have negligible I-cache MPKI.

5.2 Lean Cores and the Code They Execute

Figure 5.3 shows an intrinsic property of HPC applications: inside the parallel regions, most of the threads execute the same code. It gives the percentage of

instruction footprint shared among *all* the threads running the application. Instruction sharing is extremely high for HPC workloads. On average, around 99% of dynamically executed instructions are the same for all running threads. Different threads work on different sets of data but the same set of instructions, as in parallel loops and parallel tasks, which results in a large amount of duplication across private I-caches.

These facts motivate our study on sharing the I-cache among lean cores in an ACMP. The potential benefits include improved I-cache hit rates due to constructive cross-thread instruction prefetching, as well as savings in chip area and static power. For example, factoring out around 15% of per-core private real estate for an eight lean core cluster, opens an opportunity to spend that saving on an additional core. The main potential drawback is a larger I-cache access latency due to the introduction of a shared-access interconnection network. In this Chapter we evaluate this tradeoff and provide an optimal solution tuned to increase performance per power and area.

5.3 Shared I-cache Architecture

For a baseline configuration, we consider an ACMP composed of one large and eight lean cores with private L1 and L2 caches, connected to an on-chip memory controller giving access to off-chip memory. Figure 5.4 shows the instruction side of the baseline and proposed ACMP architectures. It presents four worker cores for simplicity. In our study, we use a configuration with one big and eight small cores. We first detail the core model, based on a decoupled front-end architecture. After that, we present the evaluated ACMP architecture with a shared I-cache among lean cores.

5.3.1 Core Front-End

Figure 5.4 shows the baseline architecture. The core model decouples the I-cache from the branch predictor with a *fetch target queue* (FTQ) [102]. With the objective of increasing fetch bandwidth, the branch predictor and FTQ work with *fetch blocks* (FB) instead of basic blocks. An FB is a sequence of instructions that

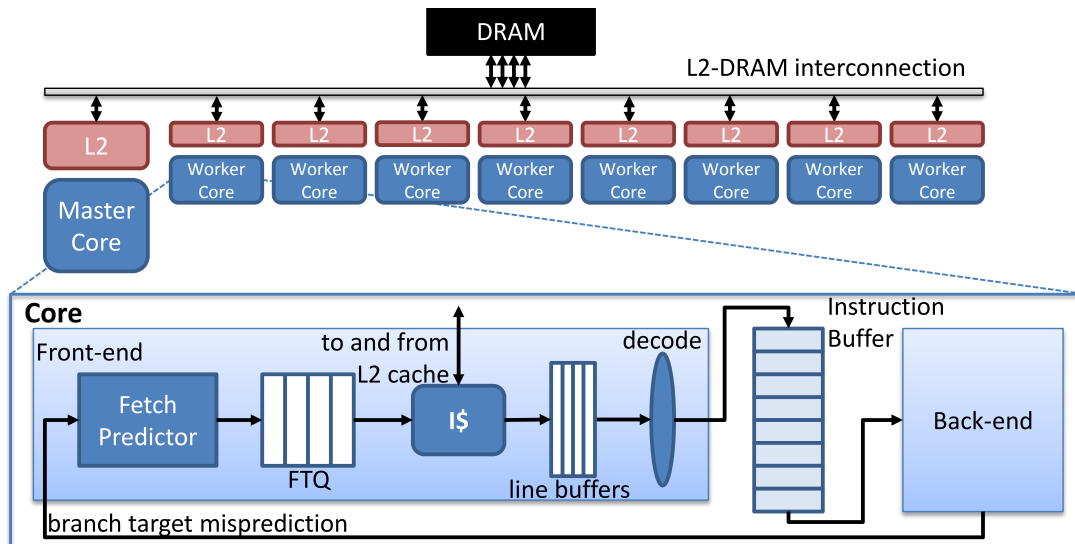


Figure 5.4: Baseline ACMP architecture with respect to the instruction part of memory hierarchy.

ends at a taken branch and, thus, it may contain multiple basic blocks if their instructions are consecutive.

The Fetch Predictor (which is actually the branch predictor) generates the fetch address for the next fetch request and stores it in the FTQ. An FTQ entry contains the starting address and the length of the FB. The private I-cache is then accessed using the FB starting address at the front of the FTQ. If the instructions to be requested to the I-cache happen to be already in one of the line buffers, no request is made to the I-cache, and the contents of that line buffer are reused instead. With more line buffers, the front-end is capable of having more outstanding requests to its I-cache, one request per line buffer. When the requested I-cache line is returned from the cache, it is stored in one of the line buffers, which act as prefetch buffers. Using shift and rotate logic, instructions are extracted from the line buffer and stored in the instruction queue. From that point, the back-end, representing the rest of the pipeline, executes and retires those instructions. In case of a branch misprediction, the pending I-cache requests are discarded and all front-end stages of the pipeline flushed.

Figure 5.5 details the shared I-cache architecture. The FB predictor, FTQ, line buffers and decode logic are as in the baseline architecture. The main difference is that the I-cache is placed outside of the core and connected to multiple cores. Depending on the sharing degree, more or less cores may share one I-cache. In

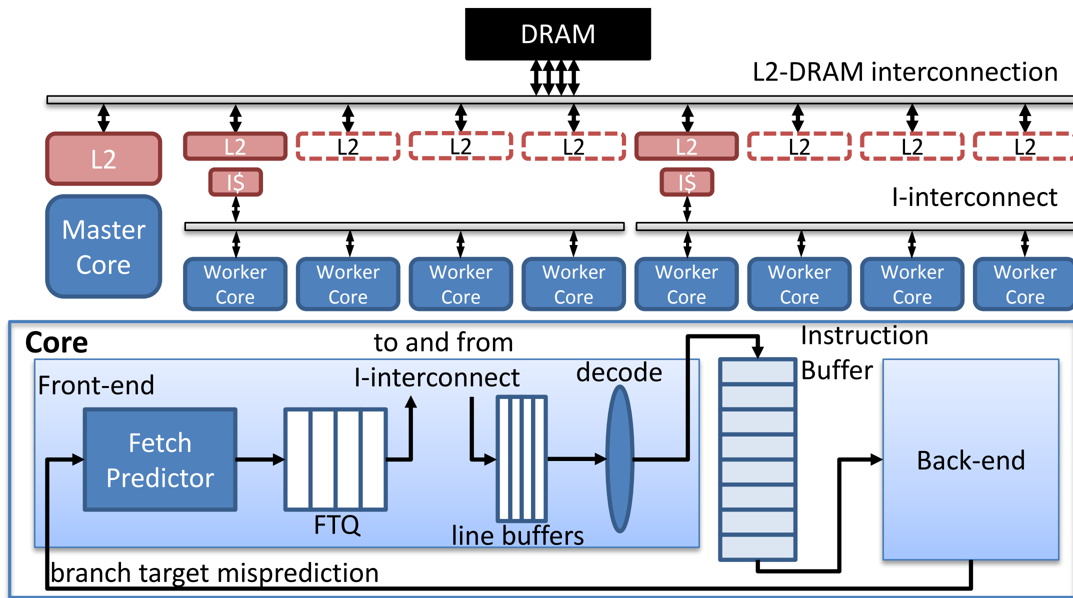


Figure 5.5: Shared I-cache ACMP architecture. Master core is not modified.

the figure, four lean cores share one I-cache thus, there are two I-caches for eight cores.

5.3.2 Shared I-cache and Interconnect

Multi-banked caches consist of several cache banks, providing multiple accesses in the same cycle, up to one access per bank. This technique is attractive for last-level caches since they are usually shared among cores. The same logic can be applied to a shared I-cache. Instead of serializing core accesses to the I-cache, multiple requests can be served as long as they fetch from different banks.

To fully utilize a multi-banked cache, all cores must be connected to all banks, which means using a crossbar switch as interconnect or multiple buses. Although crossbar and multi-buses provide higher bandwidth and reduce the congestion, they are expensive in area and power. The area cost of a crossbar increases quadratically with the number of cache banks, whereas the area of a bus increases linearly. Given our main objective of reducing area and power without hurting the performance, we evaluate this tradeoff in Section 5.5.2.

5.4 Simulation Setup

Here, we use TaskSim to model our system of interest. Table 5.1 shows the configuration parameters for the simulated ACMP. The cache hierarchy, fetch predictor, shared I-interconnect, memory controller, and off-chip memory are modeled in detail. Cores-per-cache or *cpc* stands for the number of worker cores that share one I-cache. For example, with eight worker cores in total and $cpc = 4$ there are two groups of four cores where each group share one I-cache. The I-cache size, line width, associativity and latency remain the same for any degree of sharing. We focus our evaluation on the parameters that most affect the impact of our proposal: different degrees of sharing (*cpc*), the number of line buffers, and the I-interconnect bandwidth.

We evaluate our proposal using three HPC benchmark suites: NAS Parallel Benchmarks (NPB suite), SPEC OMP 2012 (SPECOMP suite), and ExMatEx Applications. We run all of the 10 benchmarks from NPB suite with input set `C`, and 10 benchmarks from SPECOMP suite with `reference` inputs. We also use four ExMatEx Applications (`CoEVP`, `CoMD`, `CoSP`, and `LULESH`) with default input parameters. Our evaluation is based on 24 HPC workloads in total, all of them implemented using the OpenMP programming model.

We evaluate OpenMP applications in this paper but our conclusions are also applicable to other HPC programming models, including distributed memory models like MPI. Although MPI tasks run on separate processes, they still run the same executable. In such case, the OS maps all the code regions to the same physical page, since code pages are read only. The same applies for shared libraries. This means that multiple processes in MPI applications, running on a single node, share the same code as they access the same physical code pages.

5.5 Evaluation

In this section we present the evaluation of sharing the I-cache among lean cores on an ACMP. We start by checking how simple I-cache sharing affects the performance. Increasing the I-cache access latency by putting a shared bus between worker cores and the I-cache, we measure the performance loss for some workloads

Table 5.1: Configuration parameters for the simulated ACMP.

Parameter	**Value(s)**
ACMP master core worker core	1 master and 8 worker cores IPC values from an Intel's i7 core IPC values from an ARM's Cortex-A9 core
Cores-per-cache (*cpc*)	[1, 2, 4, 8] 1 stands for a baseline (private I-caches)
I-cache	size = 32 KB, 8-way latency = 1 cycle line width = 64 B
Line buffers	[2, 4, 8] width = 64 B
I$-interconnect	type = single or double bus latency = 2 cycles + contention width = 32 B arbitration = round-robin
Fetch predictor	16 KB gshare + 256-entry loop predictor
L2 cache	size = 1 MB, 32-way latency = 20 cycles line width = 64 B
L2-DRAM bus	latency = 4 cycles + contention width = 32 B
DRAM	size = unlimited timing parameters = standard

especially with the higher degrees of sharing. We evaluate how adding more line buffers and doubling the bandwidth of a shared bus overcomes this problem as a tradeoff between the performance and energy consumption. At the end, we find the scalability limits of this proposal and answer the question if a single I-cache can be shared among all cores on an ACMP, including the master core.

5.5.1 Naive I-cache Sharing

First, we evaluate sharing a 32 KB I-cache among two, four, and eight small cores, and compare with the baseline architecture (private, 32 KB I-caches). Figure 5.6 shows the normalized execution time with respect to the baseline architecture for different levels of sharing. For some benchmarks, a single I-cache shared among eight cores increases execution time, up to 18% in the case of `UA`. Figure 5.7 gives

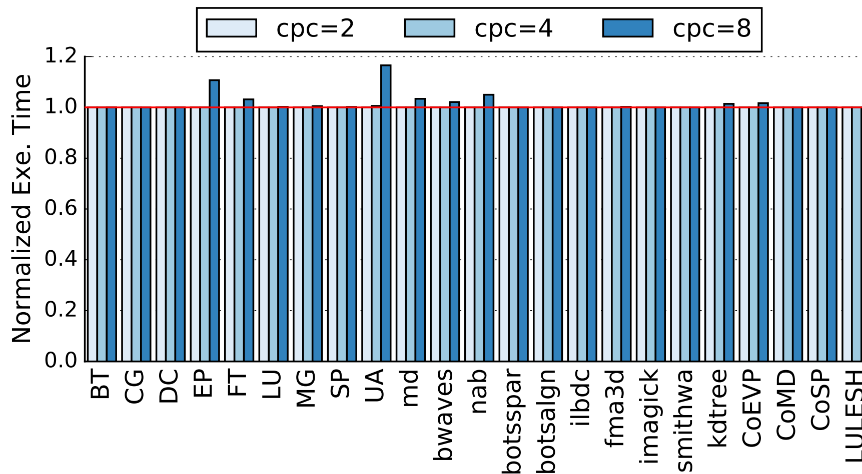


Figure 5.6: Naive scaling. Execution time for different levels of sharing a 32 KB I-cache among worker cores. We use four line buffers and a single bus as the interconnection network.

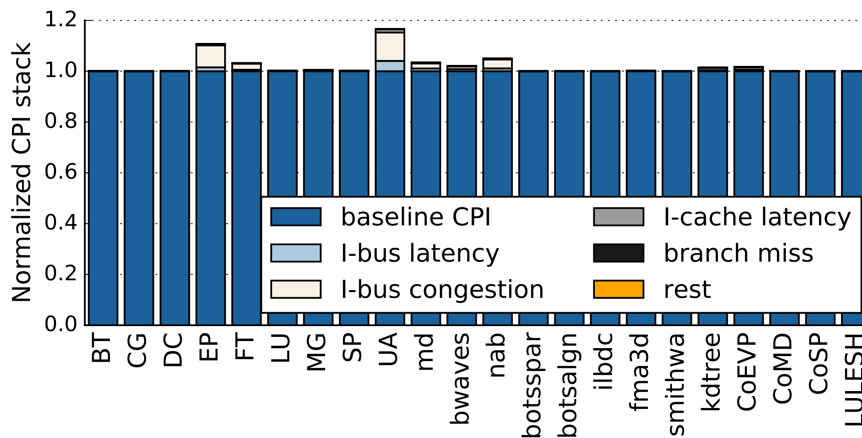


Figure 5.7: Naive scaling. Normalized CPI stack per benchmark for the highest level of sharing ($cpc = 8$).

the normalized CPI stack per benchmark when a single I-cache is shared among all eight cores. Very few additional stall cycles are caused by the latencies from I-cache misses, branch misses, and fetch requests to the upper levels of memory hierarchy. HPC applications have predictable branches and a simulated 16 KB gshare augmented with a loop predictor provides a low number of branch mispredictions (with 3.8× higher branch MPKI values in serial code than in the parallel sections). The majority of stall cycles are due to the extra latency brought by the intermediate shared bus. Most stall cycles are caused by contention on the I-bus. We explore two potential features to overcome these stall cycles: putting more line buffers or increasing the bandwidth of the shared interconnect.

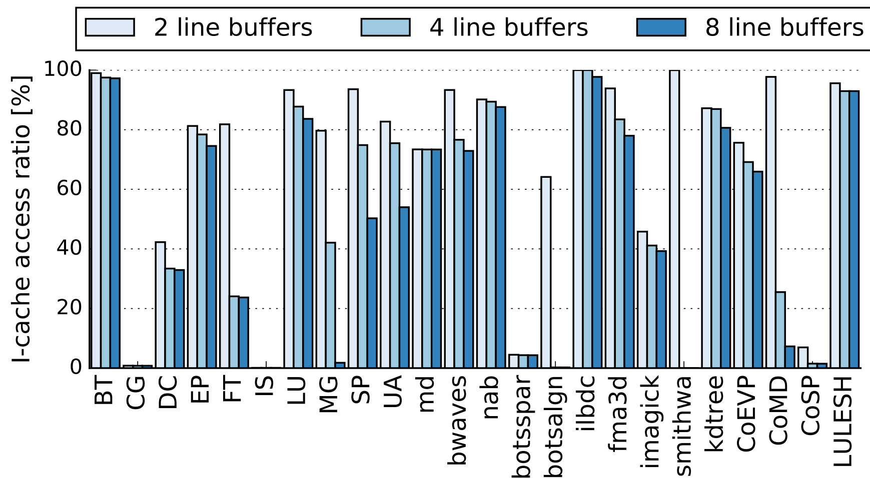


Figure 5.8: I-cache access ratio for different number of line buffers. More than eight line buffers does not reduce the I-cache access ratio significantly.

5.5.2 Scalable I-cache Sharing

With more line buffers, the front-end is capable of having more outstanding requests to its I-cache, one request per line buffer. Every time the starting address of the current fetch block exists in a line previously brought into one of the line buffers, the front-end reuses that line buffer and does not issue a request to the I-cache. This reduces the number of accesses to the shared I-cache and contention on the shared bus.

Figure 5.8 shows how using more line buffers reduces the I-cache access ratio, defined as the number of lines fetched from the I-cache divided by the total number of fetch requests. This is expected due to high temporal locality that is present in the code. It is interesting how this temporal locality complements our analysis on average basic block length (see Figure 5.1). For almost all of the benchmarks where the average basic block length is small, the I-cache access ratio is also low (`CG, IS, botsalgn, botsspar, CoSP`). On the other side, when the basic blocks are long, almost all the accesses are to the I-cache (`BT, LU, ilbdc`, and `LULESH`).

Another way of reducing the contention on a shared interconnection is to increase its bandwidth. Instead of a single bus, we use a shared multi-banked I-cache so that each bank now has its own bus connected to all worker cores. For example, having an I-cache with two banks, one with even and one with odd cache lines, we connect a separate bus for each bank, so that the I-cache requests of even cache

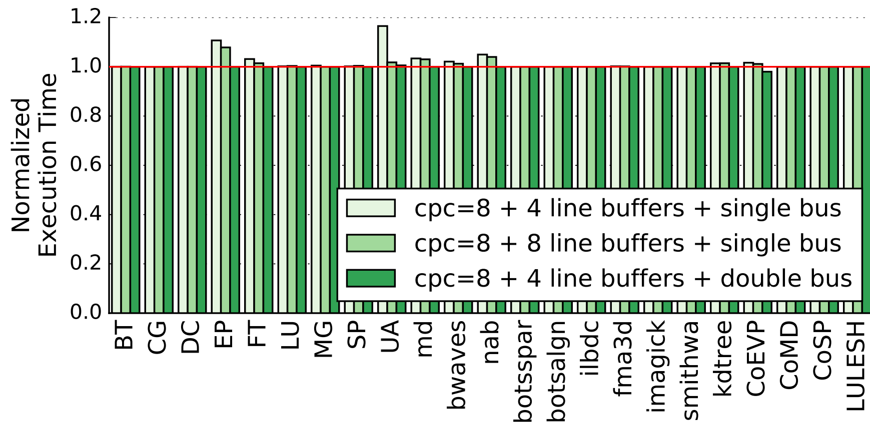


Figure 5.9: Trade-off between adding more line buffers and doubling the interconnection bandwidth when a single 16 KB I-cache is shared ($cpc = 8$). The execution times are normalized to the baseline architecture (private, 32 KB I-caches).

line addresses route through the first bus, and the requests with odd line addresses route through the second bus. That way, a shared multi-banked I-cache is able to provide two cache lines per cycle as long as they are found in different cache banks. Doubling the number of buses increases the area of the I-interconnect by 4× compared to a single bus proposal. With the cost of dedicating more area and power budget to this solution, we reduce the contention on the shared I-interconnect.

Figure 5.9 shows how these two techniques affect the total execution time. Adding more line buffers is beneficial for some benchmarks where it reduces the I-cache access ratio, such as `UA`. But, in most cases, the baseline with four line buffers already captures most executed basic blocks and hot loops, thus adding more line buffers to this set has a limited effect. On the other hand, doubling the bandwidth of the interconnection network between the lean cores and the shared I-cache completely removes the stall cycles caused by prolonged I-cache access latency. By using two I-buses instead of one, we halve the number of cores requesting the I-cache line per bus, and reduce the contention.

5.5.3 Miss Analysis

Figure 5.10 shows how sharing an I-cache among all worker threads ($cpc = 8$) affects the number of misses per kilo instruction (MPKI). The numbers above the

bars represent the absolute MPKI values obtained with a set of private, 32 KB I-caches. As we have seen before on Figure 5.2, HPC applications miss very few times accessing an I-cache in parallel regions. On average, sharing the I-cache reduces the number of misses by 50%, and up to 90% in case of `LU` and `SP`, compared to a baseline architecture (private I-caches). Even a smaller I-cache shared among all lean cores ($cpc = 8 :: 16KB$) provides fewer misses than the set of per core 32 KB I-caches. This is a direct consequence of the code sharing among threads in HPC workloads. Threads prefetch instructions for each other in a shared I-cache and we have observed in some cases *a complete absence of cold misses for some threads*. Sharing the I-cache increases the number of non-compulsory misses for some benchmarks due to the lower overall capacity (`botsalgn, smithwa`). In those cases the MPKI values are still reduced, which implies that compulsory misses are dominant. In some other cases (`SP, imagick, LULESH`), even non-compulsory misses are reduced due to almost perfect time alignment among threads accessing the same line in the shared I-cache.

The most interesting case is the `CoEVP` benchmark. That is the only HPC workload we analyse for which the I-cache MPKI value is above 1 for a private, 32 KB I-cache. Sharing a single I-cache among all worker cores halves the number of misses, and with a double I-bus we provide enough bandwidth so that congestion does not introduce additional stall cycles. With these two things combined, we even observe a 2% performance improvement, as shown on Figure 5.9. For HPC applications where I-cache misses introduce a significant performance degradation, our proposal of sharing the I-cache among lean cores stands not just as an area and power saving technique, but also to increase the performance.

5.5.4 Area and Power Savings

We estimate the area and power savings relative to a set of lean cores with private I-caches as the baseline. Master core, LLC and NoC are not included in this analysis. Sharing an I-cache among cores reduces the occupied area and total power but at the same time the additional shared bus introduces overheads.

We use McPAT [45] and CACTI [103] to estimate the area and energy consumption of cores, I-caches, I-buses, and line buffers. We have selected the ARM Cortex-A9 configuration file from the McPAT bundle because it has been validated against

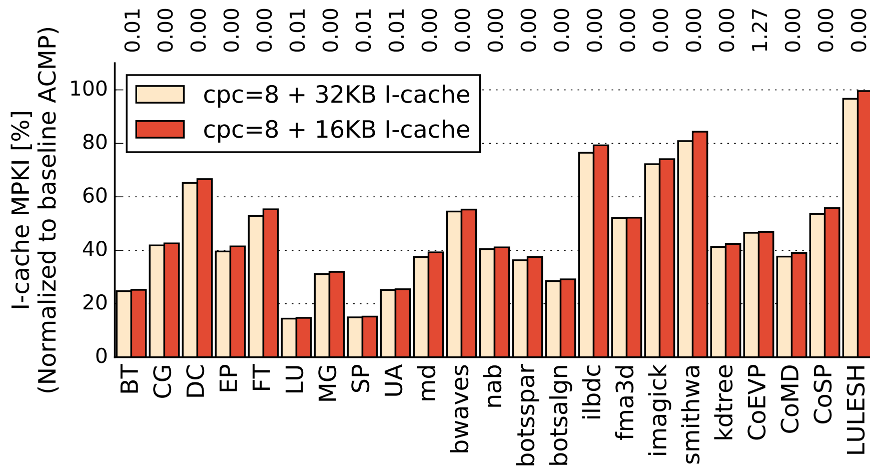


Figure 5.10: MPKI values for an I-cache shared among all eight lean cores in its two sizes, 32 KB and 16 KB, normalized to a baseline ACMP (private 32 KB I-caches). Numbers above the graph represent absolute MPKI values for each benchmark with private I-caches.

real silicon and is representative of lean cores. We run McPAT for different ACMP configurations and I-cache sizes and use statistics from simulation outputs and performance counters. Then, we obtain the area and power numbers and compare them with the baseline values.

Both wires and logic of the shared bus contribute to interconnection overhead. When a bus is wired without array structures underneath, logic can be placed under the bus without additional area overhead [104]. The area occupied by a bus is determined by the number of wires, the wire pitch and length. In our model, bus width is the same as the I-cache line width, which determines the number of wires plus address lines. The wire pitch for a 45 nm technology is 205 nm [105]. The length of the bus is estimated as the number of cores times the bus width [106]. This gives a quadratic dependence of bus area on line width. For power estimation we use the power-to-area relation taken from the McPAT values of the NoC component (bus). It gives a linear dependence of total power on area. With previously obtained area values for the bus, we apply this coefficient to get its total power numbers. For dynamic power, we set the number of transactions on the NoC as the number of accesses to the shared I-cache and apply the same dynamic-to-total power ratio, once we calculate the total power.

Figure 5.11 presents execution time, energy, and area consumption of eight worker cores for different design points, averaged across the benchmarks and normalized

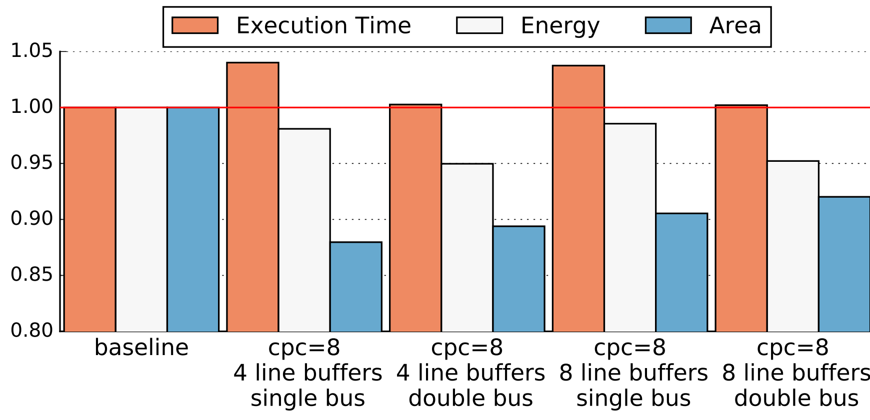


Figure 5.11: Energy and area savings adding more line buffers and doubling the interconnection bandwidth when a single 16 KB I-cache is shared ($cpc = 8$). All the values are normalized to the baseline architecture and averaged across the benchmarks.

to the baseline. For the highest level of sharing ($cpc = 8$), we focus on the trade-off between using more line buffers or doubling the bandwidth. Compared to the baseline, sharing an I-cache reduces the area and static power. The number of accesses to the shared I-cache increases 8× but since we share smaller, 16 KB I-cache, its dynamic power is also lower compared to a set of private I-caches. We calculate the energy as the product of total power (dynamic and static) and execution time. Configurations with only one I-bus have the highest area savings but modest energy savings, mostly due to the increased execution time. With the methodology explained in the previous paragraph, we estimate that the area budget of a double I-bus is around 45% of a 16 KB I-cache. More line buffers brings less activity on the bus and less accesses to the I-cache but more area and energy for a line buffer access.

Figure 5.11 also presents optimal designs for different metrics. In case we are mostly interested in area savings, sharing the I-cache among eight cores with four line buffers and single bus, stands as the optimal design. Unfortunately, it also brings 4% of performance degradation on average. If hurting the performance is not an option, the best configuration is an I-cache shared among eight lean cores with four line buffers and a double I-bus that provides savings of 5% in energy and 11% of area.

These savings can be used to increase performance for the same power and area budget. A shared I-cache architecture among worker cores allows adding an extra

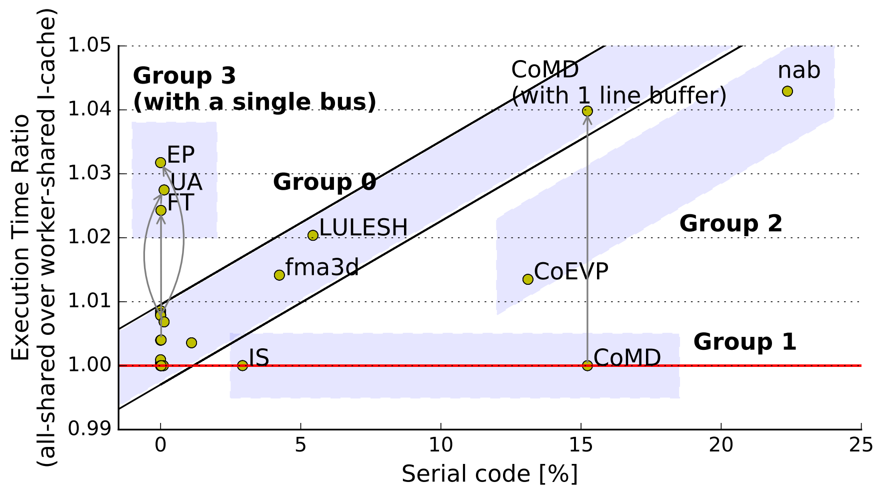


Figure 5.12: Execution time ratio dependence on the serial code fraction.

core for the same area. This can be attractive for many-core designs such as Xeon Phi, configuring the processing elements in octa-core clusters each with a single shared I-cache. Another possibility is to increase other hardware structures, such as data cache and SIMD execution unit. HPC codes benefit from additional thread- and data-level parallelism, therefore leading to higher CMP performance per unit of area and energy efficiency.

5.5.5 A Single I-cache Shared Among All Cores on an ACMP

Besides executing serial parts of the code, the master core acts as an additional worker core during parallel code sections. Here, we analyse whether the master core can be joined to the set of worker cores sharing a single I-cache. That way, it can benefit from inter-thread code prefetching and contribute further to the area and energy savings by discarding its private I-cache. In this analysis we use shared 32 KB I-cache, so that we do not hurt the master core execution by reducing the I-cache size. Configuring the I-interconnect as a double bus, we compare *all-share* proposal (master and workers share a single I-cache) with the previously evaluated *worker-shared* proposal (the I-cache is shared only among worker cores).

Figure 5.12 explains why for some benchmarks it is harmful to share a single I-cache among all cores. It shows how the performance ratio between all-shared and worker-shared proposals depends on the fraction of serial code. In general, with

higher percentage of serial code, all-shared needs more time to complete the same job compared to worker-shared configuration. The master core has more aggressive back-end (heavyweight core) and it runs alone in serial code parts that have shorter basic blocks on average. Sharing an I-cache, every time it fetches the sequential code it has to send the request through the I-interconnect bringing back fewer instructions. With the increased I-cache access latency and shorter basic blocks, the master core does not provide enough instructions to its back-end, introducing stall cycles and hurting the performance. We estimate this dependency with the area between two diagonal black lines on Figure 5.12. Still, there are few outliers that we break down into groups, each with different reasons being distant from the general dependency:

- **Group 0 - default behavior:** Most of the benchmarks belong to this group as they have a negligible amount of instructions executed in serial parts. Benchmarks like `fma3d` and `LULESH` show the general trend, for every 5% of serial code fraction the performance degrades for 1% compared to worker-shared configuration.
- **Group 1 - code locality in serial code:** Although with significant amount of instructions executed in serial by master core (especially for `CoMD`) the execution time is the same as in worker-shared setup. The reason is high code locality of serial code. For example, when we configure the core front-end with four line buffers (baseline), `CoMD` rarely accesses the shared I-cache when executing sequential code. Only one line buffer is not enough to exploit the serial code locality, thus `CoMD` moves to Group 0.
- **Group 2 - long basic blocks in serial code:** Figure 5.1 shows that HPC applications have short basic blocks in sequential code regions, with two benchmarks as exceptions, `nab` and `CoEVP`. That is the reason why these two benchmarks do not belong to Group 0. With longer basic blocks, the master core behaves like worker cores in parallel regions.
- **Group 3 - scalability limitations:** If we use a single I-bus, `EP, FT`, and `UA` benchmarks show performance degradation when the master core also shares the I-cache. This time, the stall cycles are not caused due to prolonged I-cache access latency in serial code sections, but in parallel ones. Adding one more core to a single I-bus increases the congestion and the execution

time. This finding exposes the scalability limits. Sharing an I-cache among more than eight cores introduces additional stall cycles which can not be mitigated with a double bus interconnect and four line buffers. With higher interconnection bandwidth and line buffers, the performance degradation can be reduced, but the extra area and energy cost do not justify such an investment, leading to a design with the same performance and the same area and power budget as the baseline ACMP.

This final analysis further stresses the difference between parallel code commonly run on HPC systems and serial bottleneck that exists in every parallel application. There is a need to tailor the cores on a CMP differently, depending on the parts of the code they execute. Although attractive with the additional energy and area savings, sharing an I-cache among all cores on an ACMP shows performance degradation as the amount of serial code increases. Our findings suggest that an I-cache can be shared among worker cores providing energy and area savings for the same performance, but the master core should be left with its private I-cache.

Chapter 6

Multi-socket GPU Design

The presence of accelerators in today's HPC compute nodes is a standard case given their ability to run parallel code more efficiently than general-purpose CPUs. Over the past 10 years, GPUs have become the most common compute accelerator devices not just in HPC, but also in datacenters and machine learning installations, improving the performance of many workloads beyond what Moore's Law would have predicted. They achieve high throughput and power efficiency by employing many small single instruction multiple thread (SIMT) cores, utilizing a uniform memory system and leveraging data parallelism exposed via the programming model. Future performance improvement of GPU devices depends on the growth of these SIMT core count. Still, Moore's law is slowing and while GPU die sizes have been increasing quickly over the past several generations, this growth is expected to slow down due to limitations in lithography and manufacturing costs. Without larger or denser dies, GPUs manufacturers are likely to embrace some sort of multi-GPU integration. One way seems to be already available, such in case of NVIDIA's DGX-1 compute nodes, where multiple GPUs stand as a set of pluggable devices. Another way considers closely coupled multi-socket GPU designs where transistors are more readily available. However, when moving to such designs, maintaining the illusion of a uniform memory system is increasingly difficult. In this Chapter, we first analyze the necessary runtime support in order to establish a proper baseline performance and make transparent multi-GPU executions available. After that, we investigate multi-socket non-uniform memory access (NUMA) GPU designs and show that significant changes are needed to both the GPU interconnect and cache architectures to achieve performance scalability.

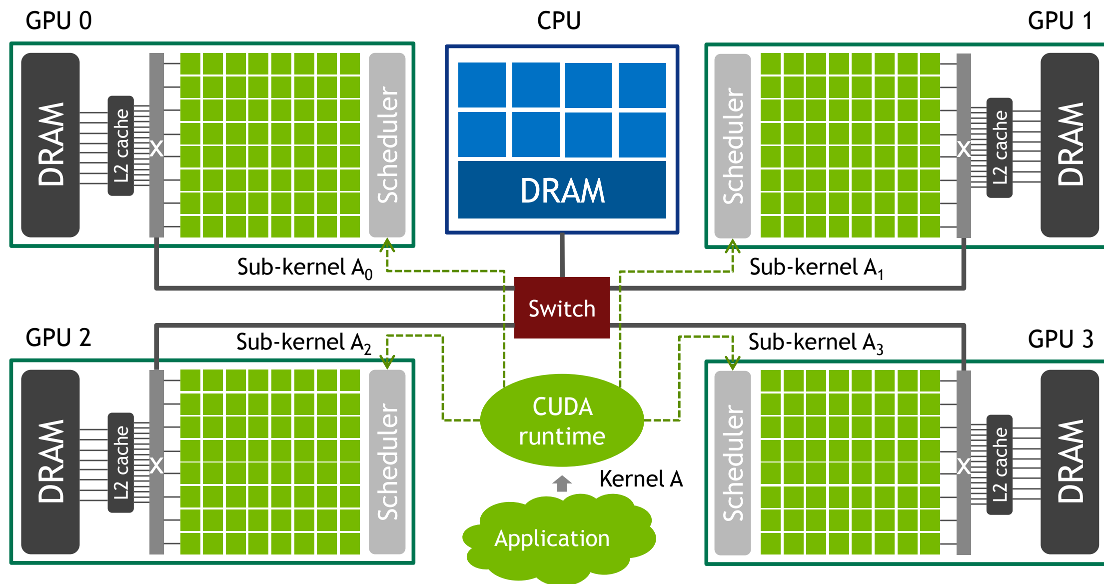


Figure 6.1: Schematic representation of proposed transparent multi-socket GPU system consisting of four GPU sockets and one CPU.

We show that application phase effects can be exploited allowing GPU sockets to dynamically optimize their individual interconnect and cache policies, minimizing the impact of NUMA effects.

6.1 System of Interest and Simulation Setup

To evaluate the performance of future NUMA-aware multi-socket GPUs we use a proprietary, cycle-level, trace-driven simulator for single and multi-GPU systems. Our baseline GPU in both single GPU and multi-socket GPU configurations, approximates the latest NVIDIA Pascal architecture [21]. Each streaming multiprocessor (SM) is modeled as an in-order processor with multiple levels of cache hierarchy containing private, per-SM, L1 caches and multi-banked, shared, L2 cache. Each GPU is backed by local on-package high bandwidth memory [72]. Our multi-socket GPU systems contain two to eight of these GPUs interconnected through a full bandwidth GPU switch as shown in Figure 6.1. Table 6.1 provides a more detailed overview of the simulation parameters. Section 3.3.2 explains the set of workloads we are using in this Chapter.

GPU coherence protocols are not one-size fits all [107–109]. This work examines clusters of large discrete GPUs but smaller more tightly integrated GPU–CPU

Table 6.1: Simulation parameters for evaluation of single and multi-socket GPU systems.

Parameter	Value(s)
Num of GPU sockets	4
Total number of SMs	64 per GPU socket
GPU Frequency	1 GHz
Max number of Warps	64 per SM
Warp Scheduler	Greedy then Round Robin
L1 Cache	Private, 128 KB per SM, 128 B lines, 4-way, Write-Through, GPU-side SW-based coherency
L2 Cache	Shared, 4 MB per socket, 128 B lines, 16-way, Write-Back, Memory-side non-coherent
GPU–GPU Interconnect	128 GB/s per socket (64 GB/s each direction) 8 lanes 8 B wide each per direction 128-cycle latency
DRAM Bandwidth	768 GB/s per GPU socket
DRAM Latency	100 ns

designs exist today as system on chips (SoC) [110, 111]. In these designs GPUs and CPUs can share a single memory space and last-level cache, necessitating a compatible GPU–CPU coherence protocol. However, closely coupled CPU-GPU solutions are not likely to be ideal candidates for GPU-centric HPC workloads. Discrete GPUs each dedicate tens of billions of transistors to throughput computing, while integrated solutions dedicate only a fraction of the chip area. While discrete GPUs are also starting to integrate more closely with some CPU coherence protocols [109, 112], PCIe attached discrete GPUs (where integrated coherence is not possible) are likely to continue dominating the market, thanks to broad compatibility between CPU and GPU vendors.

This work examines the scalability of one such cache coherence protocol used by PCIe attached discrete GPUs. The protocol is optimized for simplicity and without need for hardware coherence support at any level of the cache hierarchy. SM-side L1 private caches achieve coherence through compiler inserted cache control (flush) operations and memory-side L2 caches, which do not require coherence support. While software-based coherence may seem heavy handed compared to fine grained MOESI-style hardware coherence, many GPU programming models (in addition to C++ 2011) are moving towards scoped synchronization where explicit software acquire and release operations must be used to enforce coherence. Without the

use of these operations, coherence is not globally guaranteed and thus maintaining fine grain CPU-style MOESI coherence (via either directories or broadcast) may be an unnecessary burden.

6.2 NUMA-Aware GPU Runtime

Current GPU software and hardware is co-designed together to optimize throughput of processors based on the assumption of uniform memory properties within the GPU. Fine grained interleaving of memory addresses across memory channels on the GPU provides implicit load balancing across memory but destroys memory locality. As a result, thread block (CTA) scheduling policies need not be sophisticated to capture locality, which has been destroyed by the memory system layout. For future NUMA GPUs to work well, both system software and hardware must be changed to achieve both functionality and performance. Before focusing on architectural changes to build a NUMA-aware GPU we describe the GPU runtime system we employ to enable multi-socket GPU execution.

Prior work has demonstrated it is possible to design a framework and a runtime system that transparently decomposes GPU kernels in sub-kernels and executes them on multiple PCIe attached GPUs in parallel [80]. For example, on NVIDIA GPUs this can be implemented by intercepting and remapping each kernel call, GPU memory allocation, memory copy, and GPU-wide synchronization issued by the CUDA driver. Special care needs to ensure that per-GPU memory fences are promoted to system level and seen by all GPUs as well as guaranteeing that sub-kernel CTA identifiers are properly managed to reflect those of the original kernel. In [80] these two problems were solved by introducing code annotations and an additional source-to-source compiler which was also responsible for statically partitioning data placement and computation.

In this work, we follow a similar strategy but without using a source-to-source translation. Unlike prior work, we are able to rely on NVIDIA's Unified Virtual Addressing [29] to allow dynamic placement of pages into memory at runtime rather than static memory placement. Similarly, technologies with cache line granularity interconnects like NVIDIA's NVLink [21] allow transparent access to remote memory without the need to modify application source code to access local

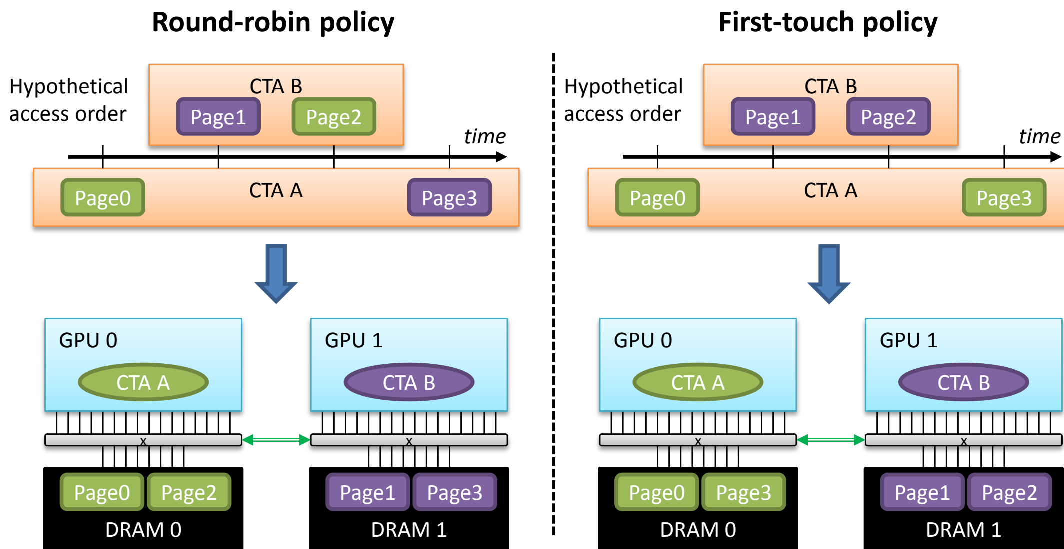
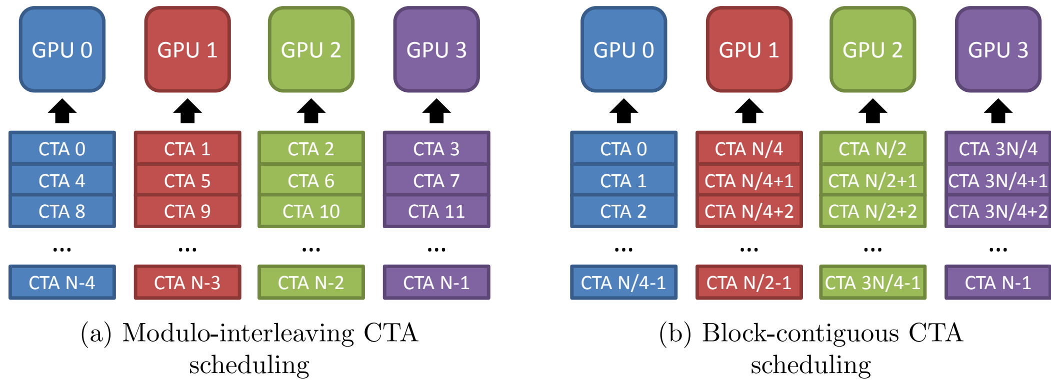


Figure 6.2: Comparison of round-robin and first-touch allocation policies on a dual-GPU system.

or remote memory addresses. Due to these advancements, we assume that through dynamic compilation of PTX to SASS at executions, the GPU runtime will be able to statically identify and promote system wide memory fences as well as manage sub-kernel CTA identifiers.

Current GPUs perform fine grained memory interleaving at a sub-page granularity across memory channels. In a NUMA GPU this policy would destroy locality and result in 75% of all accesses going to remote memory in a 4 GPU system, an undesirable effect in NUMA systems. Similarly, a round-robin page level interleaving could be utilized, like with the Linux INTERLEAVE page allocation strategy, but despite the inherent memory load balancing, this still results in 75% of memory accesses occurring over low bandwidth NUMA links. Instead we leverage UVM page migration functionality to migrate pages on-demand from system memory to local GPU memory as soon as the first access (also called first-touch allocation) is performed as described by Arunkumar et. al [113]. Figure 6.2 schematically depicts the concept and differences between round-robin and first-touch memory page allocation policies.

One way of improving the locality is to bring the data close to its accessing thread blocks through the page allocation policy. Another way is to move thread blocks close to the data through improved CTA scheduling and distribution. On a single GPU, fine grain dynamic assignment of CTAs to SMs is performed to achieve



(a) Modulo-interleaving CTA scheduling

(b) Block-contiguous CTA scheduling

Figure 6.3: Comparison of traditional and locality optimized CTA scheduling.

good load balancing. Extending this policy to a multi-socket GPU system is not possible due to the relatively high latency of passing sub-kernel launches from software to hardware. To overcome this penalty the GPU runtime must launch a block of CTAs to each GPU-socket at coarse granularity. To encourage load balancing, each sub-kernel could be comprised of an interleaving of CTAs using modulo arithmetic, such as shown on Figure 6.3a. Alternatively a single kernel can be decomposed into N sub-kernels, where N is the total number of GPU sockets in the system, assigning an equal amount of contiguous CTAs to each GPU. This design choice, presented on Figure 6.3b, potentially exposes workload unbalance across sub-kernels, but it has been shown to preserve data locality present in applications where neighboring CTAs access contiguous memory regions [80, 113].

6.2.1 Performance Through Locality

Figure 6.4 shows the relative performance of a 4-socket NUMA GPU with respect to a single GPU under the two possible CTA scheduling and memory placement strategies explained above. The green (darker) bars show the relative performance of traditional single GPU scheduling and memory interleaving policies when adapted to a NUMA GPU. The blue (lighter) bars show the relative performance of using locality optimized GPU scheduling and memory placement, consisting of contiguous block CTA scheduling and first-touch page migration. We can clearly see that the *Locality-Optimized* solution almost always outperforms the traditional

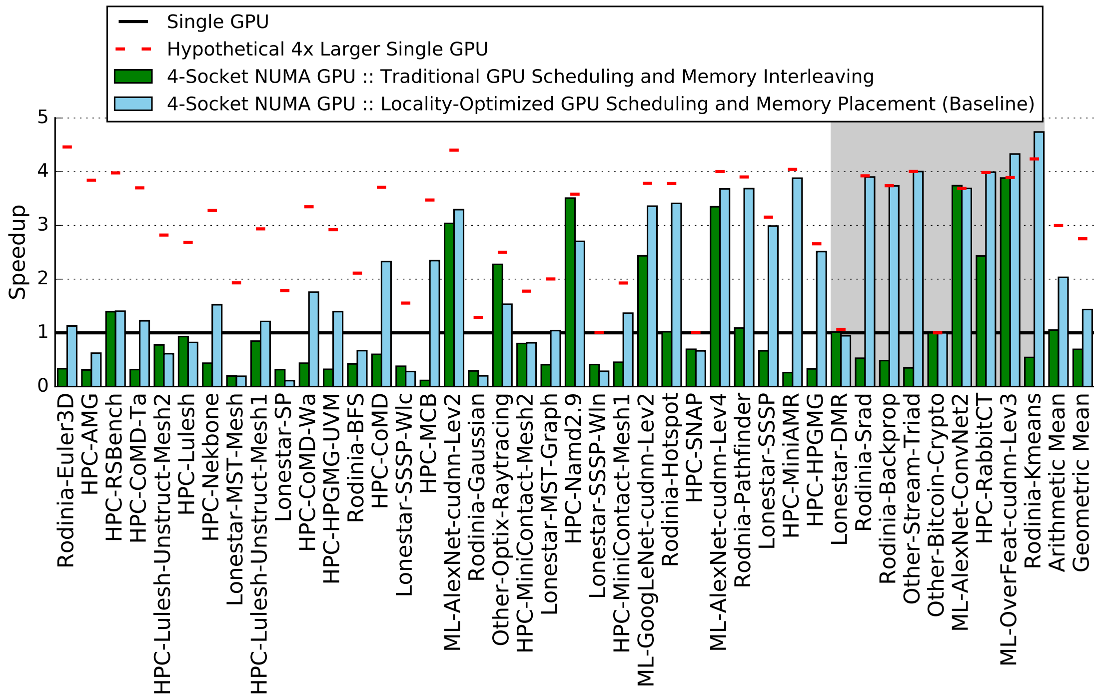


Figure 6.4: Performance of a 4-socket NUMA GPU relative to a single GPU and a hypothetical 4× larger (all resources scaled) single GPU. Applications shown in grey achieve greater than 99% of performance scaling with SW-only locality optimization.

GPU scheduling and memory interleaving. Without these runtime locality optimizations, in average a 4-socket NUMA GPU is not able to even match the performance of a single GPU despite the large increase in hardware resources. Thus from now on, using variants of prior proposals [80, 113], we only consider this locality optimized GPU runtime for the remainder of this work.

Despite the performance improvements that can come via locality optimized software runtimes, many applications do not scale well on our proposed NUMA GPU system. To illustrate this, Figure 6.4 shows the speedup achievable by a hypothetical (unbuildable) 4× larger GPU with a red dash. This red dash represent an approximation of the maximum theoretical performance we could expect from a perfectly architected (both HW and SW) NUMA GPU system. Figure 6.4 sorts the applications by the gap between relative performance of the Locality-Optimized NUMA GPU and hypothetical 4× larger GPU. We observe that on the right side of the graph some workloads (shown in the grey box) can achieve or surpass the maximum theoretical performance. In particular for the two far-most benchmarks on the right, the locality optimized solutions can outperform the hypothetical 4×

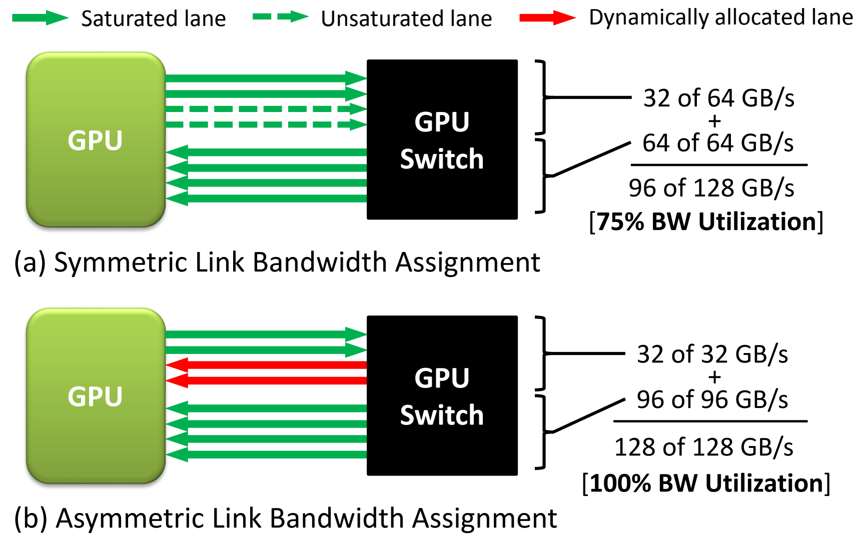


Figure 6.5: Example of dynamic link assignment to improve interconnect efficiency.

larger GPU due higher cache hitrates because contiguous block scheduling is more cache friendly than traditional GPU scheduling.

However, for the applications on the left side there is a large gap between the Locality-Optimized NUMA design and theoretical performance. These are work-loads in which either locality does not exist or the Locality-Optimized GPU run-time is not effective, resulting in large amount of remote data accesses still oc-curring. Because our goal is to provide scalable performance for single GPU opti-mized applications, in the rest of the paper we aim to close this performance gap through microarchitectural innovation. To simplify later discussion, we choose to exclude benchmarks that achieve $\geq$99% of the theoretical performance with SW-only locality optimizations. However, we include all benchmarks in our final results to show the overall performance scalability achievable with NUMA-aware multi-socket GPUs.

6.3 Asymmetric Interconnects

6.3.1 Dynamic Bandwidth Distribution

Figure 6.5(a) shows a switch connected GPU with symmetric and static link bandwidth assignment. Each link is comprised of equal number of uni-directional high-speed lanes in both directions, collectively comprising a symmetric bi-directional link. Traditional static design time link capacity assignment is very common and has several advantages. For example, only one type of I/O circuitry (egress drivers or ingress receivers) along with only one type of control logic need to be implemented at each on-chip link interface. Moreover, the multi-socket switches result in simpler designs that can easily support a statically provisioned bandwidth requirements. On the other hand, multi-socket link bandwidth utilization can have a large impact on overall system performance. Static partitioning of bandwidth, when application needs are dynamic, can leave performance on the table. Because I/O bandwidth is a limited and expensive system resource, NUMA-aware interconnects designs must look for innovations that can keep wire and I/O utilization high.

In multi-socket NUMA GPU systems, we observe that many applications have different utilization of egress and ingress channels on both a per GPU-socket basis and during different phases of execution. For example, Figure 6.6 shows a link utilization snapshot over time for `HPC-HPGMG-UVM` application running on a SW locality optimized 4-socket NUMA GPU. Vertical dotted black lines represent kernel invocations that are split across the 4 GPU-sockets. We can see that several small kernels have negligible interconnect utilization. However, for the later larger kernels, GPU0 and GPU2 fully saturate their ingress links, while GPU1 and GPU3 fully saturate their egress links. At the same time GPU0 and GPU2, and GPU1 and GPU3 are underutilizing their egress and ingress links, respectively.

In many workloads we observe one common scenario, in which all CTAs writing to the same memory range at the end of a kernel (i.e. parallel reductions, data gathering). For CTAs running on one of the sockets, GPU0 for example, these memory references are local and do not produce any traffic on the inter-socket interconnections. However CTAs dispatched to other GPUs must issue remote memory writes, saturating their egress links while ingress links remain underutilized, but

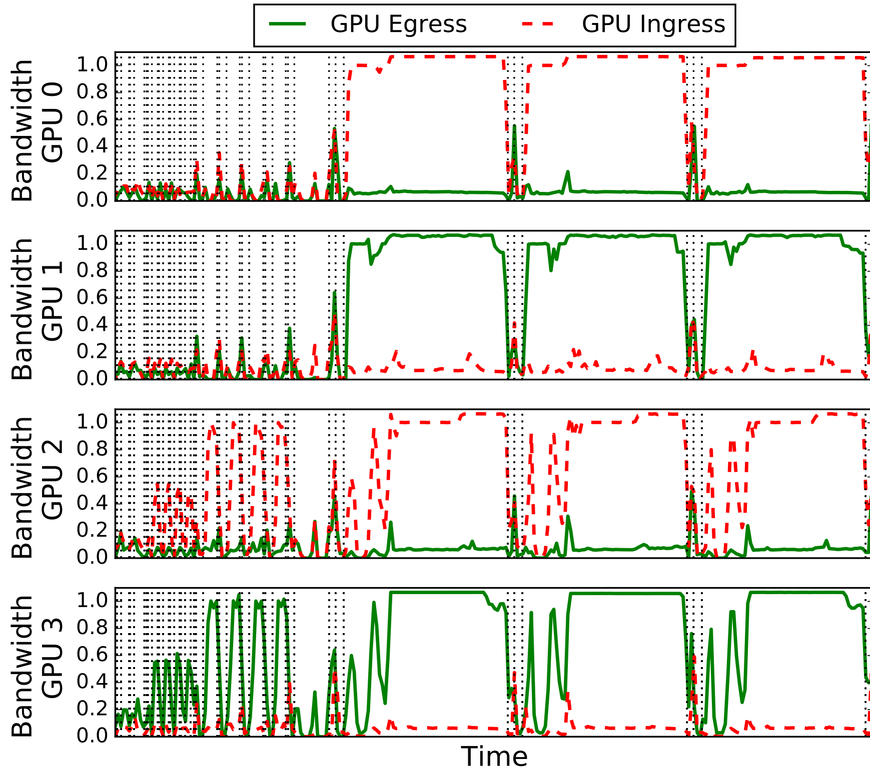


Figure 6.6: Normalized link bandwidth profile for `HPC-HPGMG-UVM` showing asymmetric link utilization between GPUs and within a GPU. Vertical black dotted lines indicate kernel launch events.

causing ingress traffic on GPU0. Such communication patterns typically utilize only 50% of available interconnect bandwidth. In these cases, dynamically increasing the number of ingress lanes for GPU0 (by turning around direction of egress lanes) and switching the direction of ingress lanes for GPUs 1–3, can substantially improve the achievable interconnect bandwidth. Motivated by these findings, we propose to dynamically control multi-socket link bandwidth assignments on a per-GPU basis resulting in dynamic asymmetric link capacity assignments as shown in Figure 6.5(b).

To evaluate this proposal we model point-to-point links containing multiple lanes, similarly to NVLink [21]. In these links, 8 lanes with 8 GB/s capacity per lane yield an aggregate bandwidth of 64 GB/s in each direction. We propose replacing uni-directional lanes with bi-directional lanes to which we apply an adaptive link bandwidth allocation mechanism that works as following. For each link in the system, at kernel launch the links are always reconfigured to contain symmetric link bandwidth with 4 lanes per direction. During kernel execution the link load

balancer periodically samples the saturation status of each link. If the lanes in one direction are not saturated, while the lanes in the opposite direction are 99% saturated, the link load balancer reconfigures and reverses the direction of one of the unsaturated lanes after quiescing all packets on that lane.

This sample and reconfigure process stops only when directional utilization is not oversubscribed or all but one lane is configured in a single direction. If both ingress and egress links are found to be saturated and in an asymmetric configuration, links are then reconfigured back towards a symmetric configuration to encourage global bandwidth equalization. While this process may sound complex, the circuitry for dynamically turning high speed single ended links around in a short number of cycles already is in use by modern high bandwidth memory interfaces such as GDDR; where the same set of wires is used for both memory reads and writes [114].

6.3.2 Results and Discussion

There are two important parameters that will affect the performance of our proposed mechanism (i) `SampleTime`: The frequency at which the scheme samples for a possible reconfiguration and (ii) `SwitchTime`: The cost of turning the direction of an individual lane. Figure 6.7 shows the performance improvement, with respect to our SW locality optimized GPU by exploring different values of the `SampleTime` indicated by green bars and assuming a `SwitchTime` of 100 cycles. The red bars in Figure 6.7 provide an upper-bound of performance speedups when doubling the available interconnect bandwidth to 256 GB/s. For workloads on the right of the figure, doubling the link bandwidth has little effect, thus dynamic link policy will also show little improvement due to low GPU–GPU interconnect bandwidth needs. On the left side, we can see that for some applications, where improved interconnect bandwidth has a large effect, dynamic lane switching can improve application performance by as much as 80%. For some benchmarks like `Rodinia-Euler-3D`, `HPC-AMG`, and `HPC-Lulesh`, doubling the link bandwidth provides 2× speedup, while our proposed dynamic link assignment mechanism is not able to significantly improve performance. Those are the workloads that saturate both link directions, so there is no opportunity to provide additional bandwidth by turning links around.

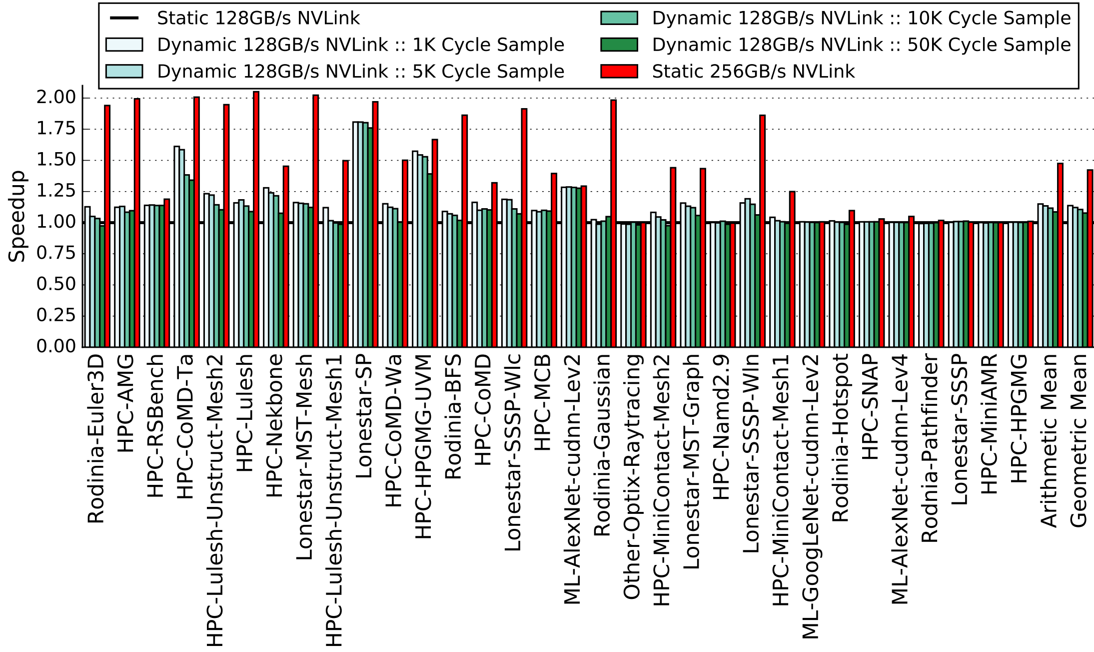


Figure 6.7: Relative speedup of the dynamic link adaptivity with respect to the baseline architecture by varying sample time and assuming switch time of 100 cycles. In red, speedup achievable by doubling link bandwidth.

Using a moderate 5 K cycle sample time (5 µs), dynamic link policy can improve performance by 14% on average over static bandwidth partitioning. If the link load balancer samples too infrequently application dynamics can be missed and performance improvement is reduced. However if the link is turned around too frequently, bandwidth is lost due to the overhead of turning the link. While we have assumed a pessimistic link turn time of 100 cycles, we performed sensitivity studies that show even if link turn time were increased to 500 cycles, our dynamic policy loses less than 2% of performance. At the same time, using a faster lane switch (10 cycles) does not significantly improve the performance over a 100 cycle link turn time. We note that the link turnaround times of modern high-speed on-board links such as GDDR5 [114] are about 8 ns including both link and internal DRAM turn-around latency (which is less than 10 cycles at 1 GHz).

Our results demonstrate that asymmetric link bandwidth allocation can be very attractive when inter-socket interconnect bandwidth is constrained by the number of on-PCB wires (and thus total link bandwidth). The primary drawback of this solution is that both types of interface circuitry (TX and RX) and logic need to be implemented for each lane in both the GPU and switch interfaces. We conducted an analysis of the potential cost of doubling the amount of I/O circuitry and logic

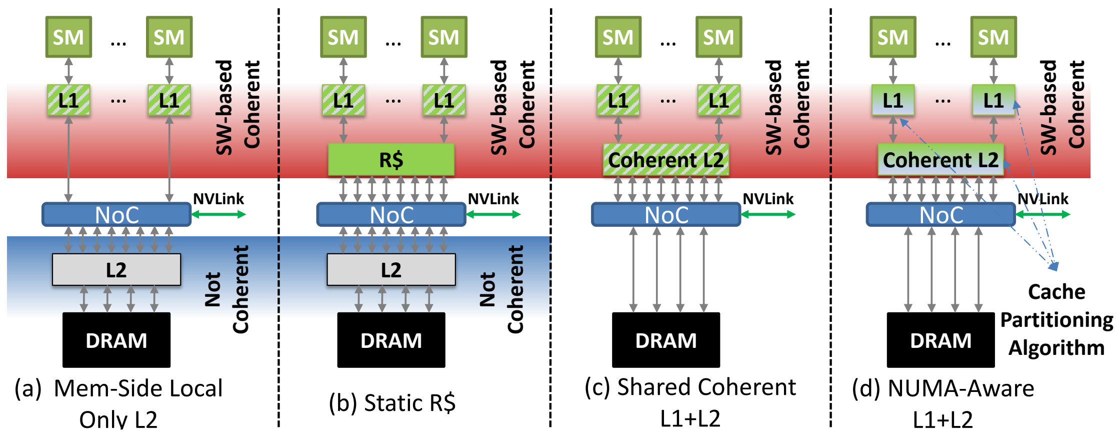


Figure 6.8: Potential L2 cache organizations to balance capacity between remote and local NUMA memory systems.

based on a proprietary state of the art GPU I/O implementation. Our results show that doubling this interface area increases total GPU area by less than 1% while yielding a 12% improvement in average interconnect bandwidth which results in a 14% application performance improvement. One additional caveat worth noting is that the proposed asymmetric link mechanism optimizes link bandwidth in a given direction for each individual link, while the total switch bandwidth remains constant.

6.4 NUMA-Aware Cache Management

In Section 6.3 we have shown that inter-socket bandwidth is an important factor in achieving scalable NUMA GPU performance. Unfortunately, because either the outgoing or incoming links must be underutilized for us to reallocate that bandwidth to the saturated link, if both incoming and outgoing links are saturated, dynamic link rebalancing yields minimal gains. To improve performance in situations where dynamic link balancing is ineffective, system designers can either increase link bandwidth, which is very expensive, or try and decrease the amount of traffic that crosses the low bandwidth communication channels. To decrease off-chip memory traffic, architects typically turn to caches to capture locality.

GPU cache hierarchies differ from traditional CPU hierarchies wherein they are not supported by strong hardware coherence protocols [115]. They also differ from CPU protocols in that caches may be both processor side (where some form

of coherence is typically necessary) or they may be memory side (where coherence is not necessary). As described in Table 6.1 and Figure 6.8(a), a GPU today is typically composed of relatively large SW managed coherent L1 caches located close to the SMs, while a relatively small, distributed, non-coherent memory side L2 cache resides close to the memory controllers. This organization works well for GPUs because their SIMT processor designs often allow for significant coalescing of requests to the same cache line, so having large L1 caches reduces the need for global crossbar bandwidth. By then placing the L2 caches memory-side they do not need to participate in the coherence protocol, reducing complexity.

6.4.1 Design Considerations

In NUMA designs remote memory references occurring across low bandwidth NUMA interconnections results in poor performance, as shown in Figure 6.4. Similarly, in NUMA GPUs utilizing traditional memory side L2 caches (that depend on fine grained memory interleaving for load balancing) is a bad decision. Because memory side caches only able to cache accesses that originate in their local memory-side, they cannot cache memory from other NUMA zones and thus can not reduce NUMA interconnect traffic. Previous work has proposed that GPU L2 cache capacity should be split between memory-side caches and a new processor-side L1.5 cache that is an extension of the GPU L1 caches [113] to enable caching of remote data, shown in Figure 6.8(b). By balancing L2 capacity between memory side and remote caches (R$), this design limits the need for extending expensive coherence operations (invalidations) into the entire L2 cache while still minimizing crossbar or interconnect bandwidth.

Flexibility: Designs that statically allocate cache capacity to local memory and remote memory, in any balance, may achieve reasonable performance in specific instances but they lack flexibility. Much like application phasing was shown to affect NUMA bandwidth consumption the ability to dynamically share cache capacity between local and remote memory has the potential to improve performance under several situations. First, when application phasing results in some GPU-sockets primarily accessing data locally while others are accessing data remotely, a fix partitioning of cache capacity is guaranteed to be sub-optimal. Second, while we show that most applications will be able to completely fill large NUMA GPUs, this

may not always be the case. GPUs within the data center are being virtualized and there is on-going work to support concurrent execution of multiple kernels within a single GPU [116, 117]. If a large NUMA GPU is sub-partitioned, it is intuitive that system software attempt to partition it along the NUMA boundaries (even within a single GPU-socket) to improve the locality of small GPU kernels. To effectively capture locality in these situation, NUMA-aware GPUs need to be able to dynamically re-purpose cache capacity at runtime, rather than be statically partitioned at design time.

Coherence: To-date, single socket GPUs have not moved their memory-side caches to processor side because the overhead of cache invalidation (due to coherence) is an unnecessary performance penalty. Within a single socket GPU with a uniform memory system, there is little performance advantage to implementing L2 caches as processor side caches. However in a multi-socket NUMA design, the performance tax of extending coherence into L2 caches is offset by the fact that remote memory accesses can now be cached locally and may be justified; Figure 6.8(c) shows a configuration with a coherent L2 cache where remote and local data contend for L2 capacity as extensions of the L1 caches, implementing identical coherence policy.

Dynamic Partitioning: Building upon coherent GPU L2 caches, we posit that while conceptually simple, allowing both remote and local memory accesses to contend for cache capacity (in both the L1 and L2 caches) in a NUMA system is flawed. In UMA systems it is well known that performance is maximized by optimizing for cache hit rate, thus minimizing off-chip memory system bandwidth. However in NUMA systems, *not all cache misses have the same relative cost performance impact*. A cache miss to a local memory address has a smaller cost (in both terms of latency and bandwidth) than a cache miss to a remote memory address. Thus, it should be beneficial to dynamically skew cache allocation to preference caching remote memory over local data when it is determined the system is bottle-necked on NUMA bandwidth.

To minimize inter-GPU bandwidth in multi-socket GPU systems we propose a NUMA-aware cache partitioning algorithm, with cache organization and brief summary shown in Figure 6.8(d) and Table 6.2. Similar to our interconnect balancing algorithm, at initial kernel launch (after GPU caches have been flushed for coherence purposes) we allocate one half of the cache ways for local memory and the

Table 6.2: Cache partitioning procedure for NUMA-aware L1 and L2 caches.

	Cache Partitioning Algorithm
Step ⓪	Allocate 1/2 ways for local and 1/2 for remote data
Step ①	Estimate NVLink incoming and monitor local DRAM outgoing BW
Step ②	If NVLink is saturated and local DRAM BW not
	RemoteWays++ and LocalWays--
Step ③	If local DRAM BW is saturated and NVLink not
	RemoteWays-- and LocalWays++
Step ④	If both are saturated
	Equalize allocated ways (++ and --)
Step ⑤	None of them is saturated
	Do nothing
Step ⑥	Go back to Step ① after `SampleTime` cycles

remaining ways for remote data (Step ⓪). After executing for a 5 K cycles period, we sample the average bandwidth utilization on local memory and estimate the GPU-socket's incoming read request rate by looking at the outgoing request rate multiplied by the response packet size. By using the outgoing request rate to estimate the incoming bandwidth, we avoid situations where incoming writes may saturate our link bandwidth falsely indicating we should preference remote data caching. Projected link utilization above 99% is considered to be bandwidth saturated (Step ①). In cases where the interconnect bandwidth is saturated but local memory bandwidth is not, the partitioning algorithm attempts to reduce remote memory traffic by re-assigning one way from the group of local ways to the remote ways grouping (Step ②). Similarly, if the local memory BW is saturated and NVLink is not, the policy re-allocates one way from the remote group, and allocates it to the group of local ways (Step ③). To minimize the impact on cache design, all ways are consulted on look up, allowing lazy eviction of data when the way partitioning changes. In case where both the interconnect and local memory bandwidth are saturated, our policy gradually equalizes the number of ways assigned for remote and local cache lines (Step ④). Finally, if neither of the links are currently saturated, the policy takes no action (Step ⑤). To prevent cache starvation of either local or remote memory (which causes memory latency dramatically increase and a subsequent drop in performance), we always require at least one way in all caches to be allocated to either remote of local memory.

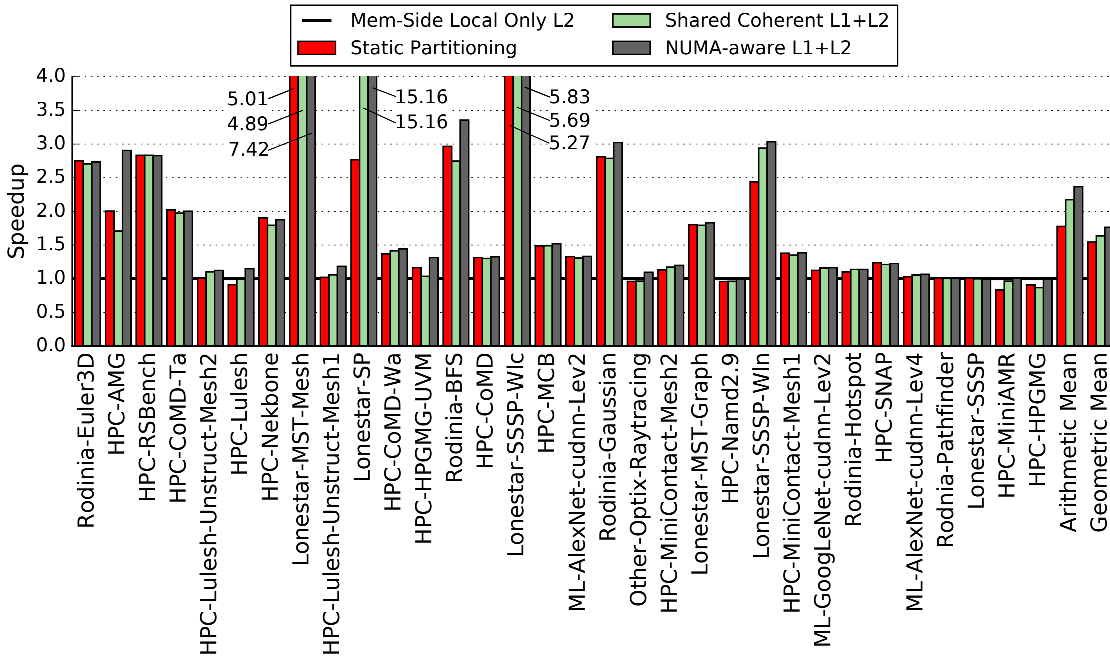


Figure 6.9: Performance of NUMA-aware dynamic cache partitioning in a 4-socket GPU compared to memory-side L2 and previously proposed static partitioning.

6.4.2 Results

Figure 6.9 compares the performance of 4 different cache configurations in our 4-socket NUMA GPU. Our baseline is a traditional GPU with memory side local-only L2 caches. To compare against prior work [113] we provide a 50–50 static partitioning where the L2 cache budget is split between the GPU-side coherent remote cache which contains only remote data, and the memory side L2 which contains only local data. In our 4-socket NUMA GPU static partitioning improves performance by 54% on average, although for some benchmarks, it hurts the performance by as much as 10% for workloads that have negligible inter-socket memory traffic. We also show the results for GPU-side coherent L1 and L2 caches where both local and remote data contend capacity. On average, this solution outperforms static cache partitioning significantly despite incurring additional flushing overhead due to cache coherence.

Finally, our proposed NUMA-aware cache partitioning policy is shown in dark grey. Due to its ability to dynamically adapt the capacity of both L2 and L1 to optimize performance when backed by NUMA memory, it is the highest performing cache configuration. By examining simulation results we find that for workloads on the

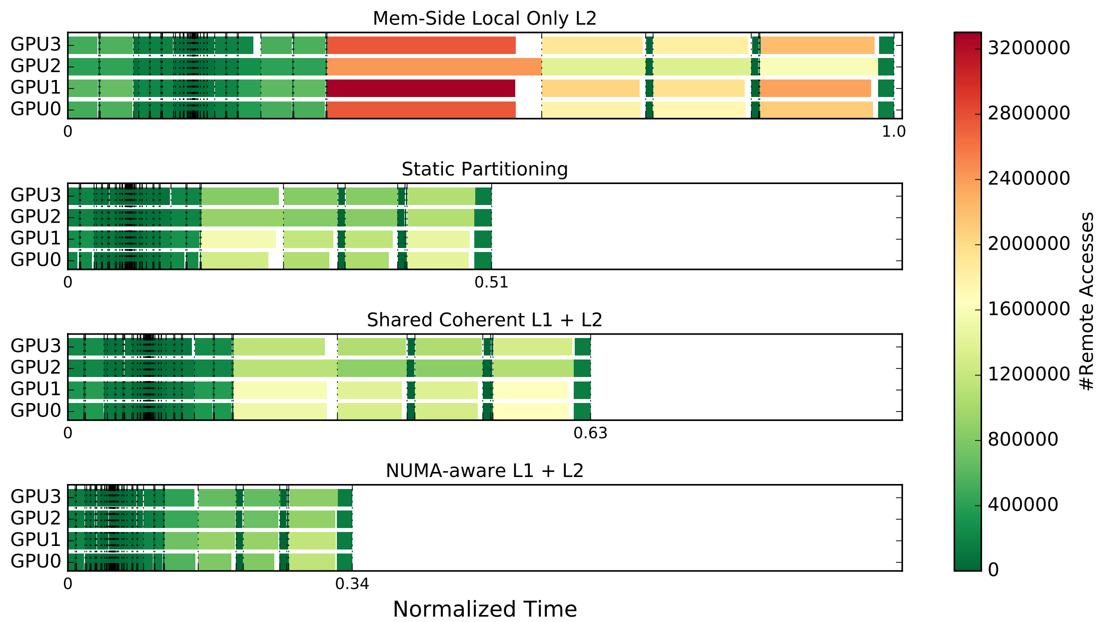


Figure 6.10: How different L2 cache organizations shown on Figure 6.8 affect the execution time in case of `HPC-AMG`. Vertical dotted lines stand for kernel launch events while colors show the number of remote accesses.

left side of Figure 6.9 which fully saturate the NVLink bandwidth, NUMA-aware dynamic policy configures the L1 and L2 caches to be primarily used as remote caches. However, workloads on the right side of the figure tend to have good GPU-socket memory locality, and thus prefer L1 and L2 caches store primarily local data. NUMA-aware cache partitioning is able to flexibly adapt to varying memory access profiles and can improve average NUMA GPU performance 76% compared to traditional memory side L2 caches, and 22% compared to previously proposed static cache partitioning despite incurring additional coherence overhead.

To visualize the effect of different cache configurations, Figure 6.10 shows per-GPU execution time of `HPC-AMG` application, normalized to memory side local-only L2 cache setup. Vertical dotted lines identify kernel launch events that are subject to sub-kernel completion requirements before the next kernel can be launched. Execution time is proportional to the number of remote accesses. Allowing L2 cache to store remote data in any kind (*Static Partitioning* or *Shared Coherent L1+L2*) reduces the pressure on the NVLink inter-socket connection links and thus, reduces the execution time. With *NUMA-aware L1+L2* dynamic cache partitioning policy, we achieve almost 3× speedup over the baseline by dedicating the entire L2 to cache remote data.

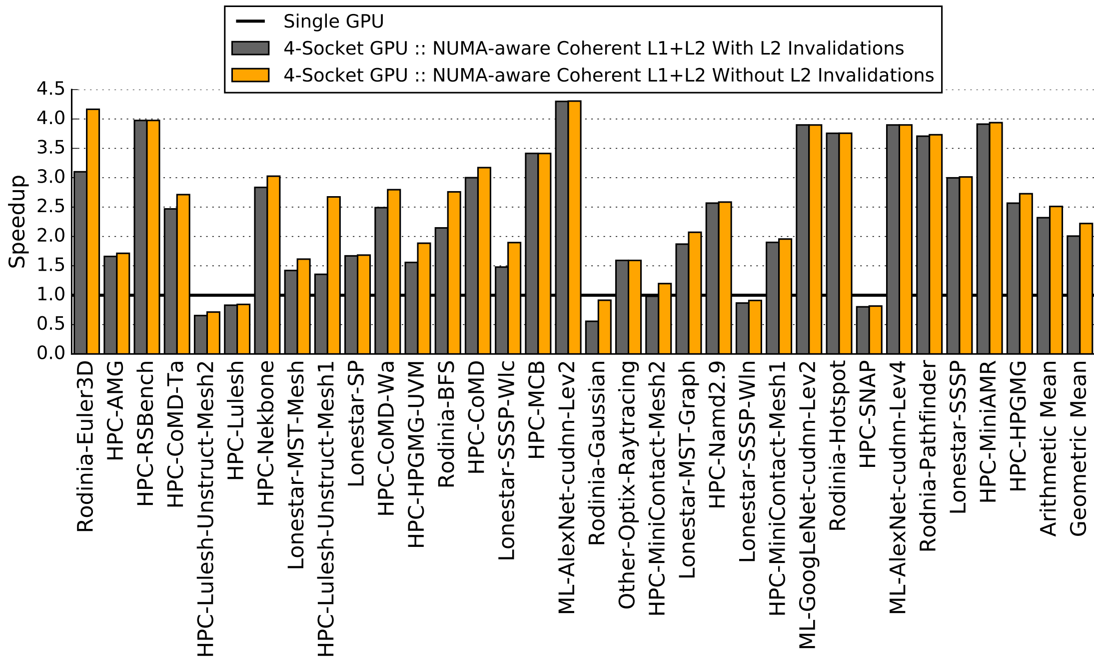


Figure 6.11: Performance overhead of extending current GPU software based coherence into the GPU L2 caches.

When extending the software controlled GPU coherence protocol into the GPU L2 caches, L1 coherence operations (flushes) must also be extended into the GPU L2 caches. To further understand the impact these coherence operations have on our NUMA-aware cache performance we evaluated a hypothetical L2 cache which need not perform these operations. Figure 6.11 shows the impact that coherence operations have on application performance in our 4-socket NUMA GPU. While significant for some applications, on average SW based GPU coherence overheads are only 10% even when extended into all GPU-socket L2 caches; we conclude that despite the coherence overheads the benefit of NUMA-aware coherent L2 caches on multi-socket GPUs is a worthy trade-off.

6.5 Discussion

6.5.1 Combined Improvement

Sections 6.3 and 6.4 provide two techniques aimed at more efficiently utilizing scarce NUMA bandwidth within future NUMA GPU systems. The proposed methods for dynamic interconnect balancing and NUMA-aware caching are orthogonal

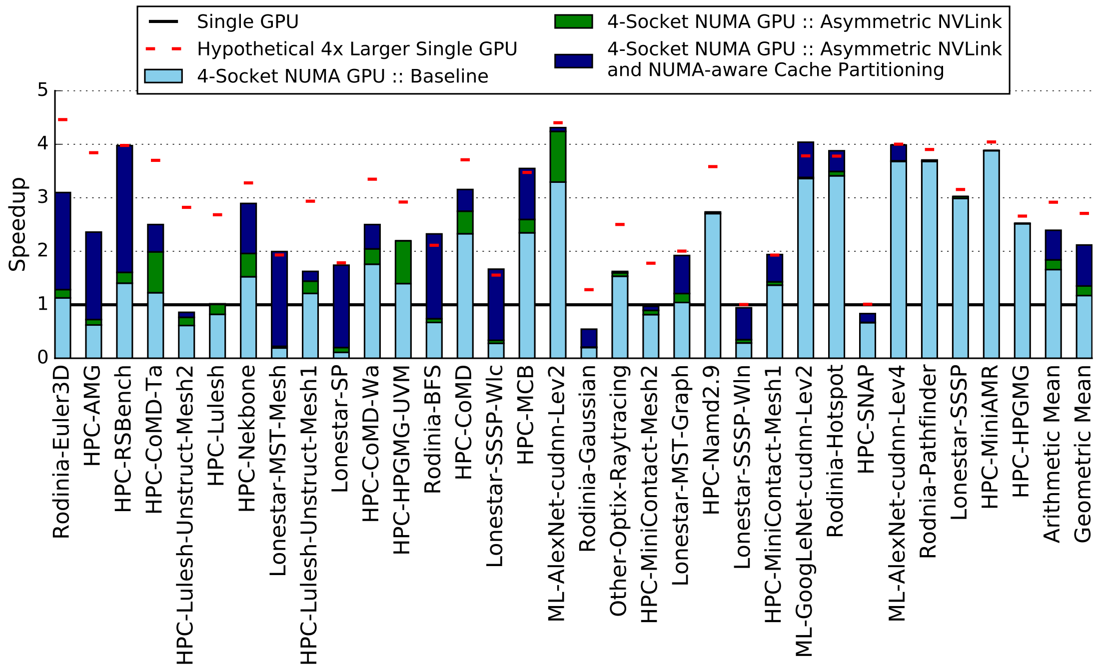


Figure 6.12: Final NUMA-aware GPU performance compared to a single GPU and 4× larger single GPU with scaled resources.

and can be applied in isolation or combination. Dynamic interconnect balancing has an implementation simplicity advantage in that the system level changes to enable this feature are isolated from the larger GPU design. Conversely, enabling NUMA-aware GPU caching based on interconnect utilization requires changes to both the physical cache architecture and the GPU coherence protocol.

Because these two features target the same problem, when employed together their effects are not strictly additive. Figure 6.12 shows the overall improvement NUMA-aware GPUs can achieve when applying both techniques in parallel. For benchmarks such as `CoMD`, these features contribute nearly equally to the overall improvement, but for others such as `ML-AlexNet-cudnn-Lev2` or `HPC-MST-Mesh1`, interconnect improvements or caching are the primary contributor respectively. On average, we observe that when combined we see 2.1× improvement over a single GPU and 80% over the baseline software locality optimized 4-socket NUMA GPU using memory side L2 caches; best performance is clearly obtained when applying both features in unison.

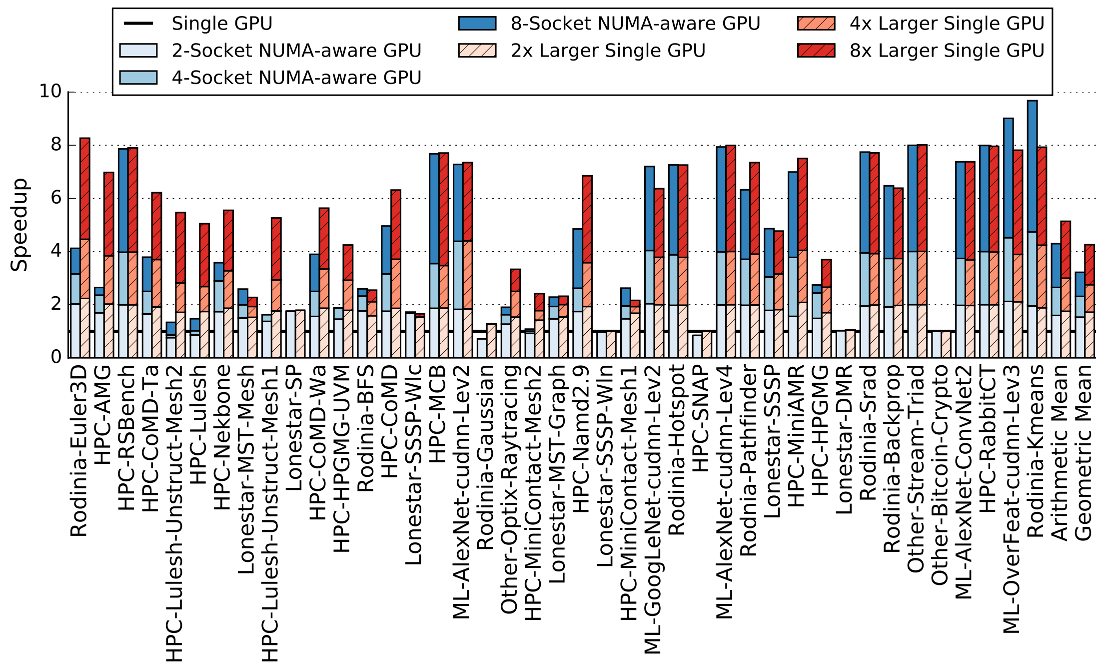


Figure 6.13: NUMA-aware 1–8 socket GPU scalability compared to hypothetical larger single GPU with scaled resources.

6.5.2 Scalability

Ultimately, for vendors to produce multi-socket NUMA GPUs they must achieve high enough parallel efficiency to justify their design. To understand the scalability of our approach Figure 6.13 shows the performance of a NUMA-aware multi-socket GPU compared to a single GPU, when scaled across 2, 4, and 8 sockets respectively. On average a 2 socket NUMA GPU achieves 1.5× speedup, while 4 sockets and 8 sockets achieve 2.3× and 3.2× speedups respectively. Depending on perspective these speedups may look attractive or lackluster; particularly when per-benchmark variance is included. However, the scalability of NUMA GPUs is not solely dependent on just NUMA GPU microarchitecture. We observe that for some applications, even if the application was run on larger hypothetical single GPUs, performance would scale similarly. This may be due to a variety of reasons beyond NUMA effects, including number of CTAs available, frequency of global synchronization, and other factors. Comparing our NUMA-aware GPU implementation to the scaling that applications could achieve on a hypothetical large single GPU, we see that NUMA-GPUs can achieve 89%, 84%, and 76% the efficiency of a hypothetical single large GPU in 2, 4, and 8 socket configurations respectively.

This high efficiency factor indicates that our design is able to largely eliminate the NUMA penalty in future multi-socket GPU designs.

6.5.3 Multi-Tenancy on Large GPUs

In this work we have shown that many workloads today have the ability to saturate (with sufficient parallel work) a GPU that is at least 8× larger than today's GPUs. With deep-data becoming commonplace across many computing paradigms, we believe that the trend of having enough parallel thread blocks to saturate large single GPUs will continue into the foreseeable future. However when GPUs become larger at the expense of having multiple addressable GPUs within the system, questions related to GPU provisioning arise. Applications that cannot saturate large GPUs will leave resources underutilized and concurrently will have to multiplex across the GPU cooperatively in time, both undesirable outcomes.

While not the focus of this work, there is significant effort in both industry and academia to support finer grain sharing of GPUs through either shared SM execution [118], spatial multiplexing of a GPU [116], or through improved time division multiplexing with GPU pre-emptability [117]. To support large GPU utilization any of these solutions could be applied to a multi-socket GPU in the cases where applications may not completely fill a larger GPU. Alternatively, with additional GPU runtime work multi-socket GPU designs could also be dynamically partitioned with a granularity of 1–N logical GPUs being exposed to the programmer, providing yet another level of flexibility to improve utilization.

6.5.4 Power Implications

As discussed earlier, arbitrarily large monolithic single GPUs are unfeasible, so multi-GPU systems connected with onboard high-speed links and switches are becoming an attractive solution for continuing GPU performance scaling. However, these onboard high-speed links and switches require additional power. We estimated the link overhead by assuming 10 pJ/b of on board interconnect energy for combined links and switch (extrapolated from publicly available information for cabinet level Mellanox switches and links [119, 120]). Using this estimate we calculate an average (Geometric Mean) 30 W of communication power for the baseline

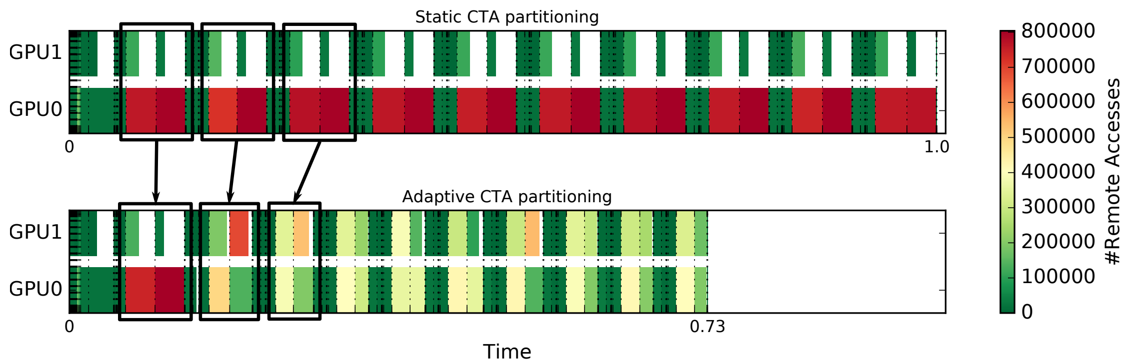


Figure 6.14: Time line of `HPC-Lulesh` with 10 time-steps executing on a dual-socket GPU. Vertical dotted lines stand for kernel launch events, and color intensity represent the number of remote memory accesses.

architecture composed of 4 GPUs, and 14 W after our NUMA-aware optimizations are applied. Some applications such as `Rodinia-Euler3D`, `HPC-Lulesh`, `HPC-AMG`, `HPC-Lulesh-Unstruct-Mesh2` are communication intensive, resulting in ≈130 W of power consumption after our optimizations are considered. Assuming a typical TDP of 250 W per GPU module, in a 4-GPU system, the extra power due to the communication represents a 5% overhead across the full range of 41 evaluated benchmarks. While this power tax is not trivial, without alternative methods for building scalable large GPUs, interconnect power will likely become a large portion of the overall GPU power budget.

6.5.5 Scheduling Improvements

Through program inspection we have identified that many workloads have a common pattern where the application launches a single GPU kernel multiple times, with only variations in parameters or input data. To improve transparent multi-socket GPU load balancing we propose exploiting this repetitive behavior by adding an adaptive heuristic to the baseline static CTA partitioning, in which the TMG runtime tracks the execution times of each issued sub-kernel. If a single kernel is then executed multiple times, the runtime will adaptively skew the consecutive CTA range assigned to each sub-kernel in an attempt to balance the individual GPU execution times.

To illustrate this effect, Figure 6.14 shows the impact of static CTA partitioning on application execution time with regard to load imbalance. For simplicity, we show a multi-socket system comprised of two GPUs. Statically splitting most

of these GPU kernels into two identically sized sub-kernels results in substantial idle time in GPU1 (shown in white); because GPU0's execution is slowed down due to a high number of remote accesses compared to GPU1 (shown in red). By dynamically balancing the number of CTAs launched to each GPU across kernel invocations, GPU1 can perform more useful computation, despite incurring more remote memory access; ultimately increasing application throughput. While more sophisticated load balancing heuristics certainly exist, we describe this simple yet effective policy, called *Adaptive CTA partitioning*, to illustrate the problem of load-imbalance in a multi-socket context.

6.5.6 Other Asymmetric Link and Cache Partitioning Proposals

Modern multi-socket CPU and GPU systems leverage advanced interconnect technologies such as NVLink, QPI and Infinity [74, 75, 121]. These modern fabrics utilize high speed serial signaling technologies over unidirectional lanes collectively comprising full-duplex links. Link capacity is statically allocated at design time and usually is symmetric in nature. In this paper we propose to dynamically re-allocate available link bandwidth resources by using same system wiring resources and on-chip I/O interfaces, while implementing both receiver and transmitter driver circuitry on each lane. This approach resembles previously proposed tristate bi-directional bus technologies [122] or former technologies such as the Intel front-side bus [123], albeit with just two bus clients. However our proposal leverages fast singled ended signaling while allowing a dynamically controlled asymmetric bandwidth allocation via on-the-fly reconfiguration of the individual lane direction within a link.

Static and dynamic cache partitioning techniques were widely explored in the context of CPU caches and QoS [124–128]. For example, Rafique et. al [126] proposed architectural support for shared cache management with quota-based approach. Qureshi et. al [127] proposed to partition cache space between applications. Jaleel et. al [128] improved on this by proposing adaptive insertion policies. Recently, cache monitoring and allocation technologies were added to Intel Xeon processors, targeted for QoS enforcement via dynamic repartitioning of on-chip CPU cache resources [125] between applications. Efficient cache partitioning in the GPU has

been explored in context of L1 caches [129] to improve application throughput. While dynamic cache partitioning has been widely used for QoS and L1 utilization, to the best of our knowledge it has never been used to try to optimize performance when caches are backed by NUMA memory systems.

Chapter 7

Conclusions

A major goal in HPC today is to develop more capable supercomputers for a given power budget. In order to reach the level of sustainable exascale computation, the fastest HPC systems today will need to improve their performance by $\sim 100\times$ and energy efficiency by $\sim 10\times$. This laudable goal requires joined effort coming from every layer of future supercomputer design.

Compute nodes have a history of evolution from single-core processing units to heterogeneous nodes with multiple accelerators. Constant growth of available transistors per chip initially introduced power-hungry CPUs optimized for sequential execution exploiting the available ILP. Reaching the limitations of single-thread performance, the following design moved to CMPs shifting the focus from ILP to TLP optimizations on the level of an entire compute node. Programming models have been supporting these new parallel architectures, exposing the existing data-parallelism within the applications. With more time spent inside the parallel code regions, computer accelerators found their use inside the supercomputers. Instead of a handful of heavyweight processors and for the same hardware budget, accelerators implement many lean cores, improving the throughput and executing the parallel code in less time. This thesis contributes to future HPC compute node organization through: better utilization of already available transistors on chip and performance improvement by aggregating multiple accelerators.

In this thesis, we found a set of valuable insights about HPC workloads regarding their requirements and effect on the core front-end hardware structures. HPC applications have fewer branch instructions, which are highly biased, and mostly

backward taken. The dynamic code footprint of HPC applications is small, and most of them fit in 16 KB. Basic blocks are long, and the distance between taken branches even longer, which enables the usage of wider I-cache lines. Branch predictors should be tailored for HPC applications and augmented with a loop branch predictor. Moreover, the results show that HPC benchmarks are not sensitive to the size of the BTB (due to the small number of branch instructions) as long as BTB associativity is high.

Compared to traditional desktop and server applications, we find that the demands of HPC applications are lower with regards to the core front-end structures. The conclusion is that the front-end is overdimensioned for these applications and, therefore, we propose a downscaling to save area and power while maintaining the same performance. This holds for cores running the parallel regions of the code but not for the one that runs the sequential sections. Our tailored core front-end configuration requires 16% less area and 7% less power in a lean-core design.

Next, our fundings presented here show the parallel code regions are executed by worker threads running the same code. In this thesis we evaluate different tradeoffs when sharing the I-cache among multiple lean cores in an ACMP. Due to initially low I-cache MPKI values and with the mutual code prefetching among threads, the shared and smaller I-cache feeds instructions to lean worker cores using a simple double bus as an I-interconnect, and a standard, small set of prefetch buffers. Our results show considerable area and energy savings of around 11% and 5%, respectively, without performance loss. The analysis suggests that constructive interference between threads reduces the number of I-cache misses and almost eliminates cold I-cache misses. In cases where the initial I-cache MPKI values were high, sharing an I-cache among worker cores even increases the performance.

Finally, this thesis tackles the problem of further performance improvement. With transistor count growth slowing and single-GPU size reaching the reticle limits, the future of scalable single GPU performance is in question. We propose that much like CPU designs have done in the past, the natural progression for continuous performance scalability of traditional GPU workloads is to move from a single to multi-socket NUMA design. This thesis shows that applying NUMA scheduling and memory placement policies inherited from the CPU world is not sufficient to achieve good performance scalability. We point out that future GPU designs will need to become NUMA-aware both in their interconnect management and

within their caching subsystems to overcome the inherent performance penalty that NUMA memory systems introduce. By leveraging software policies that preserve data locality and hardware policies that can dynamically adapt to application phases, our proposed NUMA-aware multi-socket GPU is able to outperform current GPU designs by 1.5×, 2.3×, and 3.2×, while achieving 89%, 84%, and 76% of theoretical application scalability in 2, 4, and 8 GPU sockets respectively. Our results indicate that the challenges of designing a multi-socket NUMA GPU can be solved through a combination of runtime and architectural optimization, making NUMA-aware GPUs a promising technology for scaling GPU performance beyond a single socket.

With all the major contributions presented in this thesis, we expect that future compute nodes used in HPC will continue to be heterogeneous. Tailored for single-thread performance, CPUs will be used for sequential code execution. For parallel code, we foresee the number of accelerators per compute node to be increased. Depending on the level of integration, those accelerators might be discrete pluggable devices, or tightly connected sockets, or chip modules on a single package, or any combination of those. With multiple physical memories, the runtime system has to provide the abstraction of unified virtual address space allowing users to envision such a system as a single compute device. To hide NUMA effects, microarchitectural policies have to be NUMA-aware, like cache partitioning and interconnection link distribution presented here. In case of many-core accelerators, we show how better utilization of available transistors through the shared I-cache can increase the energy-efficiency, something that GPUs already implement. With the main objectives of increasing the performance and energy efficiency, this thesis brings us a step closer to the design of future HPC systems.

7.1 Future Extensions

Serial vs. parallel code: This thesis presented the difference between sequential and parallel code regions inside HPC applications from the perspective of the core front-end structures. While we find the similar behavior between serial sections in HPC and desktop applications, we need a deeper understanding on how different this code is. Once the parallel code is offloaded to the accelerator, we need to

evaluate the serial code properties left to be executed on the master core and tailor it properly.

We also think that a similar study should be performed considering the core back-end. Evaluating the difference between serial and parallel code among HPC workloads in terms of data access pattern, prefetching, or similar, might lead to further core tailoring and improved energy-efficiency.

Extending the idea of sharing the I-cache among lean cores: Here, we show that a single smaller I-cache can be shared among up to 8 lean cores without performance degradation and with 11% of area savings. We believe that higher scalability can be achieved by evaluating more advanced fetching policies. This thesis analyzes a simple round-robin fetching policy implemented on the shared I-interconnect. Taking into account load imbalance, instruction count, or some other parameter while deciding which core has the priority when fetching the I-cache line, might potentially improve the performance.

Also, the rest of the front-end structures, such as branch predictor and branch target buffer, could be shared among the cores. Just like in case of I-cache, these structures might benefit from the mutual prefetching and training.

Other NUMA-aware policies to improve the scalability of multi-socket GPUs: In this thesis we have proposed two hardware policies to increase the performance of a multi-socket GPU. Both asymmetric link assignment and dynamic cache partitioning exploit per-application and per-GPU phase behavior, trying to overcome the bandwidth asymmetry between the local and remote memories. Although simple, we think they can be fine tuned to extract more performance. For example, if the policy detects that inter-socket links need to be rebalanced, instead of gradually turning individual links, we might speedup this process by turning a gang of links at once. Similar applies for NUMA-aware dynamic cache partitioning. For both aspects, we think that our proposals here will serve as baselines for future work on policy refining.

Next major boost of performance in transparent multi-GPU systems will come from the runtime being NUMA-aware. We already show one way to do so, through the adaptive thread block scheduling. Another is to improve the static first-touch page allocation policy, by allowing memory pages to migrate at runtime. Read-only pages that cause the significant amount of remote accesses might be replicated. We

find runtime improvements orthogonal to the microarchitectural features presented in this thesis.

7.2 Work Published

[130] **Ugljesa Milic**, Paul Carpenter, Alejandro Rico, and Alex Ramirez. *"Rebalancing the Core Front-End through HPC Code Analysis"* in proceedings of 2016 IEEE International Symposium on Workload Characterisation (IISWC 2016)

[131] **Ugljesa Milic**, Alejandro Rico, Paul Carpenter and Alex Ramirez. *"Sharing the Instruction Cache Among Lean Cores on an Asymmetric CMP for HPC Applications"* in proceedings of 2017 IEEE International Symposium on Performance Analysis of System and Software (ISPASS 2017)

[113] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, **Ugljesa Milic**, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, David Nellans. *"MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability"*, in proceedings of 2017 IEEE International Symposium on Computer Architecture (ISCA 2017)

[132] **Ugljesa Milic**, Oreste Villa, Evgeny Bolotin, Akhil Arunkumar, Eiman Ebrahimi, Aamer Jaleel, Alex Ramirez, David Nellans. *"Beyond the Socket: NUMA-Aware GPUs"*, in proceedings of 2017 IEEE/ACM International Symposium on Microarchitecture (MICRO 2017)

Abbreviations

HPC - High Performance Computing

CPU / GPU - Central / Graphics Processing Unit

NUMA / UMA - (Non-)Unified Memory Access

CMP - Chip Multiprocessor

SCMP / ACMP - Symmetric / Asymmetric CMP

FLOPS - Floating Point Operations per Second

ILP / TLP - Instruction / Thread Level Parallelism

I-cache - Instruction Cache

BP - Branch Predictor

BTB - Branch Target Buffer

MPKI - Misses per Kilo Instructions

CUDA - Compute Unified Device Architecture

SIMD - Single Instruction Multiple Data

CTA - Cooperative Thread Array

SM - Streaming Multiprocessor

SIMT - Single Instruction Multiple Threads

UVM - Unified Virtual Memory

Bibliography

[1] EXDCI. First Set of Recommendations and Reports Toward Applications. https://exdci.eu/sites/default/files/public/D3.1.pdf, 2016. [Online; accessed 2017-04-04].

[2] TOP500. Top 500 Supercomputing Cites. https://www.top500.org/lists, 2016. [Online; accessed 2017-04-04].

[3] Swiss National Supercomputing Center. Piz Daint and Piz Dora. http://www.cscs.ch/computers/piz_daint_piz_dora/index.html, 2016. [Online; accessed 2017-04-04].

[4] Paul Messina. The U.S. ExascaleComputing Project. https://exascaleproject.org/new-ecp-overview-presentation/, 2017. [Online; accessed 2017-04-04].

[5] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Journal of Parallel Computing*, 22, 1996.

[6] Timothy G Mattson and Greg Henry. An Overview of the Intel TFLOPS Supercomputer. 1998.

[7] Gordon E Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38, 1965.

[8] Tse-Yu Yeh, Deborah T Marr, and Yale N Patt. Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache. In *ACM International Conference on Supercomputing*. ACM, 2014.

[9] Thomas M Conte, Kishore N Menezes, Patrick M Mills, and Burzin A Patel. Optimization of Instruction Fetch Mechanisms for High Issue Rates. In *ACM SIGARCH Computer Architecture News*, volume 23. ACM, 1995.

[10] Chris H Perleberg and Alan Jay Smith. Branch Target Buffer Design and Optimization. *IEEE Transactions on Computers*, 42, 1993.

[11] Eric Rotenberg, Steve Bennett, and James E Smith. Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching. In *ACM/IEEE International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 1996.

[12] Robert H Dennard, Fritz H Gaensslen, Leo V Rideout, Ernest Bassous, and Andre R LeBlanc. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits*, 9, 1974.

[13] Gene M Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of Spring Joint Computer Conference*. ACM, 1967.

[14] TOP500. The Green500 List. https://www.top500.org/green500/, 2016. [Online; accessed 2017-04-04].

[15] Yuri Nishikawa, Michihiro Koibuchi, Masato Yoshimi, Kenichi Miura, and Hideharu Amano. Performance Improvement Methodology for ClearSpeed's CSX600. In *International Conference on Parallel Processing (ICPP)*. IEEE, 2007.

[16] James A Kahle, Michael N Day, H Peter Hofstee, Charles R Johns, Theodore R Maeurer, and David Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, 2005.

[17] Kevin J Barker, Kei Davis, Adolfy Hoisie, Darren J Kerbyson, Mike Lang, Scott Pakin, and Jose C Sancho. Entering the Petaflop Era: the Architecture and Performance of Roadrunner. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.

[18] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, et al. Larrabee: a Many-core x86 Architecture for Visual Computing. In *ACM Transactions on Graphics (TOG)*, volume 27, 2008.

[19] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High-performance Programming*. Newnes, 2013.

[20] Volodymyr V Kindratenko, Jeremy J Enos, Guochun Shi, Michael T Showerman, Galen W Arnold, John E Stone, James C Phillips, and Wen-mei Hwu. GPU Clusters for High-performance Computing. In *IEEE International Conference on Cluster Computing*, 2009.

[21] NVIDIA. NVIDIA Tesla P100. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016. [Online; accessed 2017-04-04].

[22] Leonardo Dagum and Ramesh Menon. OpenMP: an Industry Standard API for Shared-memory Programming. *IEEE Computational Science and Engineering*, 5, 1998.

[23] OpenMP ARB. OpenMP 4.0 Specification. http://www.openmp.org/specifications/, 2013. [Online; accessed 2017-04-04].

[24] OpenACC Organization. OpenACC 2.0 Specification. http://www.openacc.org/specification/, 2013. [Online; accessed 2017-04-04].

[25] Nvidia. CUDA C Programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2017. [Online; accessed 2017-04-04].

[26] Dean M Tullsen, Susan J Eggers, and Henry M Levy. Simultaneous Multithreading: Maximizing On-chip Parallelism. In *ACM SIGARCH Computer Architecture News*, volume 23. ACM, 1995.

[27] Debbie Marr, Frank Binns, D Hill, Glenn Hinton, D Koufaty, et al. Hyperthreading Technology in the Netburst® Microarchitecture. *14th Hot Chips*, 2002.

[28] Richard M Russell. The CRAY-1 Computer System. *Communications of the ACM*, 21, 1978.

[29] NVIDIA Corporation. Unified Memory in CUDA 6. http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/, 2013. [Online; accessed 2017-04-04].

[30] Nikolay Sakharnykh. Beyond GPU Memory Limits with Unified Memory on Pascal. https://devblogs.nvidia.com/parallelforall/beyond-gpu-memory-limits-unified-memory-pascal/, 2016. [Online; accessed 2017-04-04].

[31] ARM. Cortex-A9: Technical Reference Manual. https://www.arm.com/products/processors/cortex-m/cortex-a9.php, 2012. [Online; accessed 2017-04-04].

[32] ARM. Cortex-A15: Technical Reference Manual. https://www.arm.com/products/processors/cortex-a/cortex-a15.php, 2012. [Online; accessed 2017-04-04].

[33] Nikola Rajovic, Paul M Carpenter, Isaac Gelado, Nikola Puzovic, Alex Ramirez, and Mateo Valero. Supercomputing with Commodity CPUs: Are Mobile SoCs Ready for HPC? In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 2013.

[34] Michael Feldman. Cray to Deliver ARM-Powered Supercomputer to UK Consortium. https://www.top500.org/news/cray-to-deliver-arm-powered-supercomputer-to-uk-consortium/, 2017. [Online; accessed 2017-04-04].

[35] Norman Jouppi. Google Supercharges Machine Learning Tasks with TPU Custom Chip. *Google Blog, May*, 18, 2016.

[36] John Sell and Patrick O'Connor. The Xbox One System on a Chip and Kinect Sensor. *IEEE Micro*, 34, 2014.

[37] Dan Bouvier, Brad Cohen, Walter Fry, Sreekanth Godey, and Michael Mantor. Kabini: An AMD Accelerated Processing Unit System on a Chip. *IEEE Micro*, 34, 2014.

[38] John L Henning. SPEC CPU2006 Benchmark Descriptions. *ACM SIGARCH Computer Architecture News*, 34, 2006.

[39] Sarah Bird, Aashish Phansalkar, Lizy K John, Alex Mericas, and Rajeev Indukuru. Performance Characterization of SPEC CPU Benchmarks on Intel's Core Microarchitecture Based Processor. In *SPEC Benchmark Workshop*, 2007.

[40] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 2008.

[41] Pınar Tözün, Ippokratis Pandis, Cansu Kaynak, Djordje Jevdjic, and Anastasia Ailamaki. From A to E: analyzing TPC's OLTP Benchmarks: the Obsolete, the Ubiquitous, the Unexplored. In *Proceedings of the International Conference on Extending Database Technology*. ACM, 2013.

[42] Ruud Haring, Martin Ohmacht, Thomas Fox, Michael Gschwind, David Satterfield, Krishnan Sugavanam, Paul Coteus, Philip Heidelberger, Matthias Blumrich, Robert Wisniewski, et al. The IBM Blue Gene/Q Compute Chip. *IEEE Micro*, 2012.

[43] William J Dally, James Balfour, David Black-Shaffer, James Chen, R Curtis Harting, Vishal Parikh, Jongsoo Park, and David Sheffield. Efficient Embedded Computing. *Computer*, 41, 2008.

[44] Nvidia. NVIDIA Tegra 4 Family CPU Architecture. http://www.nvidia.com/docs/IO/116757/NVIDIA_Quad_a15_whitepaper_FINALv2.pdf, 2013. [Online: accessed 2017-04-04].

[45] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. McPAT: an Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *International Symposium on Microarchitecture (MICRO)*. IEEE, 2009.

[46] Sadaf R Alam, Richard F Barrett, Jeffery A Kuehn, Philip C Roth, and Jeffrey S Vetter. Characterization of Scientific Workloads on Systems with Multi-core Processors. In *International Symposium on Workload Characterization (IISWC)*. IEEE, 2006.

[47] Razvan Cheveresan, Matt Ramsay, Chris Feucht, and Ilya Sharapov. Characteristics of Workloads Used in High Performance and Technical Computing. In *International Conference on Supercomputing*. ACM, 2007.

[48] Prasanna Balaprakash, Darius Buntinas, Anthony Chan, Apala Guha, Rinku Gupta, Sri Hari Krishna Narayanan, Andrew A Chien, Paul Hovland, and Boyana Norris. Exascale Workload Characterization and Architecture Implications. In *Proceedings of the High Performance Computing Symposium*. International Society for Computer Simulation, 2013.

[49] Kimming So and Vittorio Zecca. Cache Performance of Vector Processors. In *ACM SIGARCH Computer Architecture News*, volume 16. ACM, 1988.

[50] Leonidas I Kontothanassis, Rabin A Sugumar, GJ Faanes, James E Smith, and Michael L Scott. Cache Performance in Vector Supercomputers. In *ACM/IEEE Conference on Supercomputing*, 1994.

[51] David J Kuck, Paul P Budnik, Shyh-Ching Chen, Duncan H Lawrie, Ross A Towle, Richard E Strebendt, Edward W Davis, Joseph Han, Paul W Kraska, and Yoichi Muraoka. Measurements of Parallelism in Ordinary FORTRAN Programs. *Computer*, 7, 1974.

[52] Avinash Sodani. Knights Landing (KNL): 2nd Generation Intel® Xeon Phi Processor. In *IEEE Hot Chips 27 Symposium (HCS)*. IEEE, 2015.

[53] Takumi Maruyama, Toshio Yoshida, Ryuji Kan, Iwao Yamazaki, Shuji Yamamura, Noriyuki Takahashi, Mikio Hondou, and Hiroshi Okano. Sparc64 VIIIfx: A New-generation Octocore Processor for Petascale Computing. *IEEE Micro*, 30, 2010.

[54] Manish Arora, Siddhartha Nath, Subhra Mazumdar, Scott B Baden, and Dean M Tullsen. Redefining the Role of the CPU in the Era of CPU-GPU Integration. *IEEE Micro*, 32, 2012.

[55] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying GPU Microarchitecture Through Microbenchmarking. In *IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, 2010.

[56] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28, 2008.

[57] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the Clouds: a Study of Emerging Scale-out Workloads on Modern Hardware. In *ACM SIGPLAN Notices*, volume 47. ACM, 2012.

[58] Jag Bolaria. Cortex-A57 Extends ARM's Reach. *Microprocessor Report*, 2012.

[59] Arunmoezhi Ramachandran, Jerome Vienne, Rob Van Der Wijngaart, Lars Koesterke, and Ilya Sharapov. Performance Evaluation of NAS Parallel Benchmarks on Intel Xeon Phi. In *International Conference on Parallel Processing (ICPP)*. IEEE, 2013.

[60] Mark D Hill and Michael R Marty. Amdahl's Law in the Multicore Era. *Computer*, 41, 2008.

[61] Andrew Danowitz, Kyle Kelley, James Mao, John P Stevenson, and Mark Horowitz. Cpu DB: Recording Microprocessor History. *Communications of the ACM*, 55, 2012.

[62] Dean M Tullsen, Susan J Eggers, Joel S Emer, Henry M Levy, Jack L Lo, and Rebecca L Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *ACM SIGARCH Computer Architecture News*. ACM, 1996.

[63] Rakesh Kumar, Norman P Jouppi, and Dean M Tullsen. Conjoined-core Chip Multiprocessing. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2004.

[64] Romain Dolbeau and André Seznec. CASH: Revisiting Hardware Sharing in Single-chip Parallel Processor. 2002.

[65] George Almási, Călin Caşcaval, Jose G Castanos, Monty Denneau, Derek Lieber, José E Moreira, and Henry S Warren Jr. Dissecting Cyclops: A Detailed Analysis of a Multithreaded Architecture. *ACM SIGARCH Computer Architecture News*, 2003.

[66] Michael Butler, Leslie Barnes, Debjit Das Sarma, and Bob Gelinas. Bulldozer: An Approach to Multithreaded Compute Performance. *IEEE Micro*, 2011.

[67] Partha Kundu, Murali Annavaram, Trung Diep, and John Shen. A Case for Shared Instruction Cache on Chip Multiprocessors Running OLTP. In *ACM SIGARCH Computer Architecture News*. ACM, 2003.

[68] Daniele Bortolotti, Francesco Paterna, Christian Pinto, Andrea Marongiu, Martino Ruggiero, and Luca Benini. Exploring Instruction Caching Strategies for Tightly-coupled Shared-memory Clusters. In *International Symposium on System on Chip (SoC)*. IEEE, 2011.

[69] John Nickolls and William J Dally. The GPU Computing Era. *IEEE Micro*, 2010.

[70] David Kanter. Knights Landing Reshapes HPC. *Microprocessor Report*, 2015.

[71] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. http://www.nvidia.es/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009. [Online; accessed 2017-04-04].

[72] JEDEC. High Bandwidth Memory(HBM) DRAM - JESD235. http://www.jedec.org/standards-documents/results/jesd235, 2015. [Online; accessed 2017-04-04].

[73] Jouke Verbree, Erik Jan Marinissen, Philippe Roussel, and Dimitrios Velenis. On the Cost-Effectiveness of Matching Repositories of Pre-tested Wafers for Wafer-to-Wafer 3D Chip Stacking. In *IEEE European Test Symposium*, 2010.

[74] NVIDIA. The World's First AI Supercomputer in a Box. http://www.nvidia.com/object/deep-learning-system.html. [Online; accessed 2017-04-04].

[75] INTEL Corporation. An Introduction to the Intel QuickPath Interconnect. http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html, 2009. [Online; accessed 2017-04-04].

[76] HyperTransport Consortium. HyperTransport 3.1 Specification. http://www.hypertransport.org/ht-3-1-link-spec, 2010. [Online; accessed 2017-04-04].

[77] Broadcom. PCI Express Switches. https://www.broadcom.com/products/pcie-switches-bridges/pcie-switches/, 2017. [Online; accessed 2017-04-04].

[78] KHRONOS GROUP. OpenCL 2.2 API Specification (Provisional). https://www.khronos.org/opencl/, 2016. [Online; accessed 2017-04-04].

[79] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. Transparent CPU-GPU Collaboration for Data-Parallel Kernels on Heterogeneous Systems. In *International Conference on Parallel architectures and compilation techniques (PACT)*. IEEE Press, 2013.

[80] Javier Cabezas, Lluís Vilanova, Isaac Gelado, Thomas B. Jablin, Nacho Navarro, and Wen-mei W. Hwu. Automatic Parallelization of Kernels in Shared-Memory Multi-GPU Nodes. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS, 2015.

[81] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *ACM SIGPLAN notices*, volume 40. ACM, 2005.

[82] Trevor E Carlson, Wim Heirmant, and Lieven Eeckhout. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-core Simulation. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2011.

[83] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 Simulator. *SIGARCH Computer Architecture News*, 39, 2011.

[84] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P Jouppi. CACTI 5.1. Technical report, Technical Report HPL-2008-20, HP Labs, 2008.

[85] Jim Turley. Cortex-A15 "Eagle" flies the coop. *Microprocessor Report*, November 2010.

[86] Baruch Solomon, Avi Mendelson, Ronny Ronen, Doron Orenstien, and Yoav Almog. Micro-operation Cache: A Power Aware Frontend for Variable Instruction Length ISA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2003.

[87] Alejandro Rico, Alejandro Duran, Felipe Cabarcas, Yoav Etsion, Alex Ramirez, and Mateo Valero. Trace-driven Simulation of Multithreaded Applications. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2011.

[88] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R Johnson, David Nellans, Mike O'Connor, and Stephen W Keckler. Flexible Software Profiling of GPU Architectures. In *ACM International Symposium on Computer Architecture (ISCA)*, volume 43. ACM, 2015.

[89] Timothy C Germann, Allen L McPherson, James F Belak, and David F Richards. Exascale Co-Design Center for Materials in Extreme Environments. http://www.exmatex.org/proxy-over.html, 2013.

[90] Vishal Aslot, Max Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B Jones, and Bodo Parady. SPEC OMP: A New Benchmark Suite for Measuring Parallel Computer Performance. In *International Workshop on OpenMP Applications and Tools*. Springer, 2001.

[91] Hao-Qiang Jin, Michael Frumkin, and Jerry Yan. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. 1999.

[92] CORAL Benchmarks. https://asc.llnl.gov/CORAL-benchmarks/, 2014. [Online; accessed 2017-04-04].

[93] M. A. O'Neil and M. Burtscher. Microarchitectural Performance Characterization of Irregular GPU Kernels. In *International Symposium on Workload Characterization (IISWC)*, 2014.

[94] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. October 2009.

[95] Sanjay J Patel, Tony Tung, Satarupa Bose, and Matthew M Crum. Increasing the Size of Atomic Instruction Blocks Using Control Flow Assertions. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2000.

[96] Alex Ramirez, Luiz André Barroso, Kourosh Gharachorloo, Robert Cohn, Josep Larriba-Pey, P Geoffrey Lowney, and Mateo Valero. Code Layout

Optimizations for Transaction Processing Workloads. In *ACM SIGARCH Computer Architecture News*, volume 29. ACM, 2001.

[97] Glenn Reinman, Brad Calder, and Todd Austin. Fetch Directed Instruction Prefetching. In *International Symposium on Microarchitecture (MICRO)*. IEEE, 1999.

[98] Scott McFarling. Combining Branch Predictors. Technical report, Technical Report TN-36, Digital Western Research Laboratory, 1993.

[99] Richard E Kessler, Edward J McLellan, and David A Webb. The Alpha 21264 Microprocessor Architecture. In *International Conference on Computer Design (ICCD)*. IEEE, 1998.

[100] André Seznec and Pierre Michaud. A Case for (partially) TAgged GEometric History Length Branch Prediction. *Journal of Instruction Level Parallelism*, 8, 2006.

[101] André Seznec. The L-TAGE Branch Predictor. *Journal of Instruction-Level Parallelism*, 2007.

[102] Glenn Reinman, Todd Austin, and Brad Calder. A Scalable Front-end Architecture for Fast Instruction Delivery. In *ACM SIGARCH Computer Architecture News*. IEEE Computer Society, 1999.

[103] Premkishore Shivakumar and Norman P Jouppi. Cacti 3.0: An Integrated Cache Timing, Power, and Area Model. Technical report, Technical Report 2001/2, Compaq Computer Corporation, 2001.

[104] Rakesh Kumar, Victor Zyuban, and Dean M Tullsen. Interconnections in Multi-core Architectures: Understanding Mechanisms, Overheads and Scaling. In *International Symposium on Computer Architecture (ISCA)*. IEEE, 2005.

[105] Junghee Lee, Chrysostomos Nicopoulos, Sung Joo Park, Madhavan Swaminathan, and Jongman Kim. Do We Need Wide Flits in Networks-on-chip? In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2013.

[106] Smaïl Niar, Lieven Eeckhout, and Koenraad De Bosschere. Comparing Multiported Cache Schemes. In *PDPTA*, 2003.

[107] Matthew D Sinclair, Johnathan Alsop, and Sarita V Adve. Efficient GPU Synchronization Without Scopes: Saying No to Complex Consistency Models. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015.

[108] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. Heterogeneous System Coherence for Integrated CPU-GPU Systems. December 2013.

[109] Amir Kavyan Ziabari, Yifan Sun, Yenai Ma, Dana Schaa, José L Abellán, Rafael Ubal, John Kim, Ajay Joshi, and David Kaeli. UMH: A Hardware-Based Unified Memory Hierarchy for Systems with Multiple Discrete GPUs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2016.

[110] Benjamin Munger, David Akeson, Srikanth Arekapudi, Tom Burd, Harry R Fair, Jim Farrell, Dave Johnson, Guhan Krishnan, Hugh McIntyre, Edward McLellan, et al. Carrizo: A High Performance, Energy Efficient 28 nm APU. *Journal of Solid-State Circuits*, 2016.

[111] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake. *IEEE Micro*, 2017.

[112] Neha Agarwal, David Nellans, Eiman Ebrahimi, Thomas F Wenisch, John Danskin, and Stephen W Keckler. Selective GPU Caches to Eliminate CPU-GPU HW Cache Coherence. In *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016.

[113] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability. In *International Symposium on Computer Architecture (ISCA)*. IEEE, 2017.

[114] Hynix. Hynix GDDR5 SGRAM Part H5GQ1H24AFR Datasheet Revision 1.0. https://www.skhynix.com/eolproducts.view.do?pronm=GDDR5+SDRAM&srnm=H5GQ1H24AFR&rk=26&rc=graphics, 2009. [Online; accessed 2017-04-04].

[115] Inderpreet Singh, Arrvindh Shriraman, Wilson W. L. Fung, Mike O'Connor, and Tor M. Aamodt. Cache Coherence for GPU Architectures. In *International Symposium on High-Performance Computer Architecture (HPCA)*, February 2013.

[116] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. *ACM SIGARCH Computer Architecture News*, 2015.

[117] Zhen Lin, Lars Nyland, and Huiyang Zhou. Enabling Efficient Preemption for SIMT Architectures with Lightweight Context Switching. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2016.

[118] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling Preemptive Multiprogramming on GPUs. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2014.

[119] Switch-IB 2 EDR Switch Silicon - World's First Smart Switch. http://www.mellanox.com/related-docs/prod_silicon/PB_SwitchIB2_EDR_Switch_Silicon.pdf, 2015. [Online; accessed 2017-04-04].

[120] ConnectX-4 VPI Single and Dual Port QSFP28 Adapter Card User Manual. http://www.mellanox.com/related-docs/user_manuals/ConnectX-4_VPI_Single_and_Dual_QSFP28_Port_Adapter_Card_User_Manual.pdf, 2016. [Online; accessed 2017-04-04].

[121] AMD. AMD's Infinity Fabric Detailed. http://wccftech.com/amds-infinity-fabric-detailed/, 2017. [Online; accessed 2017-04-04].

[122] John R. Spence and Michael M. Yamamura. Clocked Tri-State Driver Circuit. https://www.google.com/patents/US4504745, 1985.

[123] Intel. Intel Xeon Processor with 533 MHz Front Side Bus at 2 GHz to 3.20 GHz. http://download.intel.com/support/processors/xeon/sb/25213506.pdf. [Online; accessed 2017-04-04].

[124] J. Chang and G. S. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. *Proceedings of International Supercomputing Conference (ISC)*, June 2007.

[125] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer. Cache QoS: From Concept to Reality in the Intel Xeon Processor E5-2600 v3 Product Family. *International Symposium on High-Performance Computer Architecture (HPCA)*, 2016.

[126] N. Rafique, W.T. Lim, and M. Thottethodi. Architectural Support for OS-driven CMP Cache Management. *Proceedings of Parallel Architectures and Compilation Techniques (PACT)*, Sep 2006.

[127] M. Qureshi and Y. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. *International Symposium on Microarchitecture (MICRO)*, Dec 2006.

[128] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, Jr S. Steely, and J. Emer. Adaptive Insertion Policies for Managing Shared Caches. *International Symposium on Microarchitecture (MICRO)*, Oct 2008.

[129] D. Li, M. Rhu, D. R. Johnson, M. O'Connor, M. Erez, D. Burger, D. S. Fussell, and S. W. Keckler. Priority-Based Cache Allocation in Throughput Processors. *International Symposium on High-Performance Computer Architecture (HPCA)*, 2015.

[130] Ugljesa Milic, Paul Carpenter, Alejandro Rico, and Alex Ramirez. Rebalancing the Core Front-end Through HPC Code Analysis. In *IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2016.

[131] Ugljesa Milic, Alejandro Rico, Paul Carpenter, and Alex Ramirez. Sharing the Instruction Cache Among Lean Cores on an Asymmetric CMP for HPC Applications. In *IEEE International Symposium on Performance Analysis of System and Software (ISPASS)*. IEEE, 2017.

[132] Ugljesa Milic, Oreste Villa, Evgeny Bolotin, Akhil Arunkumar, Eiman Ebrahimi, Aamer Jaleel, Alex Ramirez, and David Nellans. Beyond the Socket: NUMA-Aware GPUs. In *International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 2017.