The Performance Potential for Single Application Heterogeneous Systems*

Henry Wong
University of Toronto
henry@eecg.utoronto.ca

Tor M. Aamodt
University of British Columbia
aamodt@ece.ubc.ca

Abstract

*A consideration of Amdahl's Law [9] suggests a single-chip multiprocessor with asymmetric cores is a promising way to improve performance [16]. In this paper, we conduct a limit study of the potential benefit of the tighter integration of a fast sequential core designed for instruction level parallelism (e.g., an out-of-order superscalar) and a large number of smaller cores designed for thread-level parallelism (e.g., a graphics processor). We optimally schedule instructions across cores under assumptions used in past ILP limit studies. We measure sensitivity to the sequential performance (instruction read-after-write latency) of the low-cost parallel cores, and latency and bandwidth of the communication channel between these cores and the fast sequential core. We find that the potential speedup of traditional "general purpose" applications (e.g., those from SpecCPU) as well as a heterogeneous workload (game physics) on a CPU+GPU system is low (2.2× to 12.7×), due to poor sequential performance of the parallel cores. Communication latency and bandwidth have comparatively small performance impact (*1.07× *to* 1.48×*) calling into question whether integrating onto one chip both an array of small parallel cores and a larger core will, in practice, benefit the performance of these workloads significantly when compared to a system using two separate specialized chips.*

1 Introduction

As the number of cores integrated on a single chip continues to increase, the question of how useful additional cores will be is of intense interest. Recently, Hill and Marty [16] combined Amdahl's Law [9] and Pollack's Rule [28] to quantify the notion that single-chip asymmetric multicore processors may provide better performance than using the same silicon area for a single core or some number of identical cores. In this paper we take a step towards refining this analysis by considering real workloads and their behavior scheduled on an idealized machine while modeling communication latency and bandwidth limits.

Heterogeneous systems typically use a traditional microprocessor core optimized for extracting *instruction level parallelism* (ILP) for serial tasks, while offloading parallel sections of algorithms to an array of smaller cores to efficiently exploit available data and/or thread level parallelism. The Cell processor [10] is a heterogeneous multicore system, where a traditional PowerPC core resides on the same die as an array of eight smaller cores. Existing GPU compute systems [22, 2] typically consist of a GPU with a discrete GPU attached via a card on a PCI Express bus. Although development of CPU-GPU single-chip systems has been announced [1], there is little published information quantifying the benefits of such integration.

One common characteristic of heterogeneous multicore systems employing GPUs is that the small multicores for exploiting parallelism are unable to execute a single thread of execution as fast as the larger sequential processor in the system. For example, recent GPUs from NVIDIA have a register to register read-after-write latency equivalent to 24 shader clock cycles [25][1]. This latency is due in part to the use of fine grained multithreading [32] to hide memory access and arithmetic logic unit latency [13]. Our limit study is designed to capture this effect.

While there have been previous limit studies on parallelism in the context of single-threaded machines [7, 17, 15], and homogeneous multicore machines [21], a heterogeneous system presents a different set of trade-

*Work done while the first author was at the University of British Columbia.

[1]The CUDA programming manual indicates 192 threads are required to hide read-after-write latency within a single thread, there are 32-threads per warp, and each warp is issued over four clock cycles.

offs. It is no longer merely a question of how much parallelism can be extracted, but also whether the parallelism is sufficient considering the lower sequential performance (higher register read-after-write latency) and communication overheads between processors. Furthermore, applications with sufficient thread-level parallelism to hide communication latencies may diminish the need for a single-chip heterogeneous system except where system cost considerations limit total silicon area to that available on a single-chip.

This paper makes the following contributions:

- We perform a limit study of an optimistic heterogeneous system consisting of a sequential processor and a parallel processor, modeling a traditional CPU and an array of simpler cores for exploiting parallelism. We use a dynamic programming algorithm to choose points along the instruction trace where mode switches should occur such that the total runtime of the trace, including the penalties incurred for switching modes, is minimized.

- We show the parallel processor array's *sequential performance* (read-after-write latency) relative to the performance of the sequential processor (CPU core) is a significant limitation on achievable speedup for a set of general-purpose applications. Note this is *not* the same as saying performance is limited by the serial portion of the computation [9].

- We find that latency and bandwidth between the two processors have comparatively minor effects on speedup.

In the case of a heterogeneous system using a GPU-like parallel processor, speedup is limited to only 12.7× for SPECfp 2000, 2.2× for SPECint 2000, and 2.5× for PhysicsBench [31]. When connecting the GPU using an off-chip PCI Express-like bus, SPECfp achieves 74%, SPECint 94%, and PhysicsBench 82% of the speedup achievable without latency and bandwidth limitations.

We present our processor model in Section 2, methodology in Section 3, analyze our results in Section 4, review previous limit studies in Section 5, and conclude in Section 6.

2 Modeling a Heterogeneous System

We model heterogeneous systems as having two processors with different characteristics (Figure 1). The *sequential processor* models a traditional processor core optimized for ILP, while the *parallel processor* models an array of cores for exploiting thread-level parallelism. The parallel processor models an array of low-cost cores by allowing parallelism, but with a longer register read-after-write latency than the sequential processor. The two processors may communicate over a communication channel whose latency is high and bandwidth is limited when the two processors are on separate chips. We assume that the processors are attached to ideal memory systems. Specifically, for dependencies between instructions within a given core (sequential or parallel) we assume store-to-load communication has the same latency as communication via registers (register read-after-write latency) on the same core. Thus, the effects of long latency memory access for the parallel core (assuming GPU-like fine-grained multi-threading to tolerate cache misses) is captured in the long read-to-write delay. The effects of caches and prefetching on the sequential processor core are captured by its relatively short read-to-write delay. We model a single-chip system (Figure 1(a)) with shared memory by only considering synchronization latency, potentially accomplished via shared memory (❶) and on-chip coherent caches [33]. We model a system with private memory (Figure 1(b)) by limiting the communication channel's bandwidth and imposing a latency when data needs to be copied across the link (❷) between the two processors.

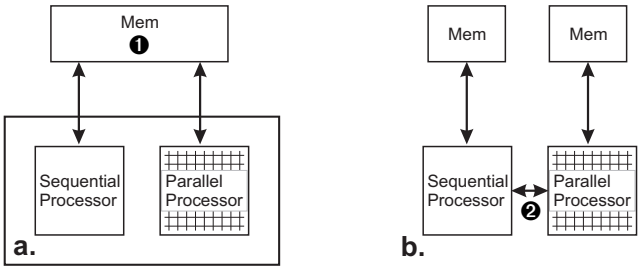

Figure 1. Conceptual Model of a Heterogeneous System. Two processors with different characteristics (a) may, or (b) may not share memory.

Section 2.1 and 2.2 describe each portion of our model in more detail. In Section 2.3 we describe our algorithm for partitioning and scheduling an instruction trace to optimize its runtime on the sequential and parallel processors.

2.1 Sequential Processor

We model the sequential processor as being able to execute one instruction per cycle (CPI of one). This simple model has the advantage of having predictable performance characteristics that make the optimal scheduling (Section 2.3) of work between sequential and parallel processors feasible. It preserves the

essential characteristic of high-ILP processors that a program is executed serially, while avoiding the modeling complexity of a more detailed model. Although this simple model does not capture the CPI effects of a sequential processor which exploits ILP, we are mainly interested in the relative speeds between the sequential and parallel processors. We account for sequential processor performance due to ILP by making the parallel processor relatively slower. In the remainder of this paper, all time periods are expressed in terms of the sequential processor's cycle time.

2.2 Parallel Processor

We model the parallel processor as a dataflow processor, where a data dependency takes multiple cycles to resolve. This dataflow model is driven by our trace based limit study methodology described in Section 3.2, which assumes perfectly predicted branches to uncover parallelism. Using a dataflow model, we avoid the requirement of partitioning instructions into threads, as done in the thread-programming model. This allows us to model the upper bound of parallelism for future programming models that may be more flexible than threads.

The parallel processor can execute multiple instructions in parallel, provided data dependencies are satisfied. Slower sequential performance of the parallel processor is modeled by increasing the latency from the beginning of an instruction's execution until the time its result is available for a dependent instruction. We do not limit the parallelism that can be used by the program, as we are interested in the amount of parallelism available in algorithms.

Our model can represent a variety of methods of building parallel hardware. In addition to an array of single-threaded cores, it can also model cores using fine-grain multithreading, like current GPUs. Note that modern GPUs from AMD [3] and NVIDIA [23, 24] provide a *scalar* multithreaded programming abstraction even though the underlying hardware is single-instruction, multiple data (SIMD). This execution mode has been called *single-instruction, multiple thread* (SIMT) [14].

In GPUs, fine-grain multithreading creates the illusion of a large amount of parallelism (>10,000s of threads) with low per-thread performance, although physically there is a lower amount of parallelism (100s of operations per cycle), high utilization of the ALUs, and frequent thread switching. GPUs use the large number of threads to "hide" register read-after-write latencies and memory access latencies by switching to a ready thread. From the perspective of the algorithm, a GPU appears as a highly-parallel, low-sequential-performance parallel processor.

To model current GPUs, we use a register read-after-write latency of 100 cycles. For example, current Nvidia GPUs have a read-after-write latency of 24 shader clocks [25] and a shader clock frequency of 1.3-1.5 GHz [23, 24]. The 100 cycle estimates includes the effect of instruction latency (24×), the difference between the shader clock and current CPU clock speeds (about 2×), and the ability of current CPUs to extract ILP—we assume an average IPC of 2 on current CPUs, resulting in another factor of 2×. We ignore factors such as SIMT branch divergence [8].

We note that the SPE cores on the Cell processor have comparable read-after-write latency to the more general purpose PPE core. However, the SPE cores are not optimized for control-flow intensive code [10] and thus may potentially suffer a higher "effective" read-after-write latency on some general purpose code (although quantifying such effects is beyond the scope of this work).

2.3 Heterogeneity

We model a heterogeneous system by allowing an algorithm to choose between executing on the sequential processor or parallel processor and to switch between them (which we refer to as a "mode switch"). We do not allow concurrent execution of both processors. This is a common paradigm, where a parallel section of work is spawned off to a co-processor while the main processor waits for the results. The runtime difference for optimal concurrent processing (e.g., as in the asymmetric multicore chips analysis given by Hill and Marty [16]) is no better than 2× compared to not allowing concurrency.

We schedule an instruction trace for alternating execution on the two processors. Execution of a trace on each type of core was described in Sections 2.1 and 2.2. For each mode switch, we impose a "mode switch cost", intuitively modeling synchronization time during which no useful work is performed. The mode switch cost is used to model communication latency and bandwidth as described in Sections 2.3.2 and 2.3.3, respectively. Next we describe our scheduling algorithm in more detail.

2.3.1 Scheduling Algorithm

Dynamic Programming is often applied to find optimal solutions to optimization problems. The paradigm requires that an optimal solution to a problem be recursively decomposable into optimal solutions of smaller

sub-problems, with the solutions to the sub-problems computed first and saved in a table to avoid recomputation [6].

In our dynamic programming algorithm, we aim to compute the set of mode switches (i.e., scheduling) of the given instruction trace that will minimize execution time, given the constraints of our model. We decompose the optimal solution to the whole trace into sub-problems that are optimal solutions to shorter traces with the same beginning, with the ultimate sub-problem being the trace with only the first instruction that can be trivially scheduled. We recursively define a solution to a longer trace by observing that a solution to a long trace is composed of a solution to a shorter sub-trace, followed by a decision on whether to perform a mode switch, followed by execution of the remaining instructions in the chosen mode.

The dynamic programming algorithm keeps a N×2 state table when given an input trace of N instructions. Each entry in the state table records the cost of an optimal scheduling for every sub-trace (N of them) and mode that was last used in those sub-traces (2 modes). At each step of the algorithm, a solution for the next sub-trace requires examining all possible locations of the previous mode switch to find the one that gives the best schedule. For each possible mode switch location, the corresponding entry of the state table is examined to retrieve the optimal solution for the sub-trace that executes all instructions up to that entry in the corresponding state (execution on the sequential, or the parallel core, respectively). This value is used to compute a candidate state table entry for the current step by adding the mode switch cost (if switching modes), and the cost to execute the remaining section of the trace from the candidate switch point up to the current instruction in the current mode (sequential, parallel). The lowest cost candidate over all earlier sub-traces is chosen for the current sub-trace.

The naive optimal algorithm described above runs in quadratic time with respect to the instruction trace length. For traces of millions of instructions in length, quadratic time is too slow. We make an approximation to enable the algorithm to run in time linear in the length of the instruction trace. Instead of looking back at all past instructions for each potential mode switch point, we only look back 30,000 instructions. The modified algorithm is no longer optimal. We mitigate this sub-optimality by first reordering instructions before scheduling. We observed that the amount of sub-optimality using this approach is insignificant.

To overcome the limitation of looking back only 30,000 instructions in our algorithm, we reorder instructions in dataflow order before scheduling. Dataflow order is the order in which instructions would execute if scheduled with our optimal scheduling algorithm. This linear-time preprocessing step exposes parallelism found anywhere in the instruction trace by grouping together instructions that can execute in parallel.

We remove instructions from the trace that do not depend on the result of any other instruction. Most of these instructions are dead code created by our method of exposing loop- and function-level parallelism, described in Section 3.2. Since dead code can execute in parallel, we remove these instructions to avoid having them inflate the amount of parallelism we observe. Across our benchmark set, 27% of instructions are removed by this mechanism. Note that the dead code we are removing is not necessarily dynamically dead [5], but rather overhead related to sequential execution of parallel code. The large number of instructions removed results from, for example, the expansion of x86 push and pop instructions (for register spills/fills) into a load or store micro-op (which we keep) and a stack-pointer update micro-op (which we do not keep).

2.3.2 Latency

We model the latency of migrating tasks between processors by imposing a constant runtime cost for each mode switch. This cost is intended to model the latency of spawning a task, as well as transferring of data between the processors. If the amount of data transferred is large relative to the bandwidth of the link between processors, this is not a good model for the cost of a mode switch. This model is reasonable when the mode switch is dominated by latency, for example in a heterogeneous multicore system where the memory hierarchy is shared (Figure 1(a)), so very little data needs to be copied between the processors.

As described in Section 2.3, our scheduling algorithm considers the cost of mode switches. A mode switch cost of zero would allow freely switching between modes, while a very high cost would constrain the scheduler to choose to run the entire trace on one processor or the other, whichever was faster.

2.3.3 Bandwidth

Bandwidth is a constraint that limits the rate that data can be transferred between processors in our model. Note that this does not apply to the processors' link to its memory (Figure 1), which we assume to be unconstrained. In our shared-memory model (Figure 1(a)) mode switches do not need to copy large amounts of data so only latency (Section 2.3.2) is a relevant constraint. In our private-memory model (Figure 1(b)),

bandwidth is consumed on the link connecting processors as a result of a mode switch.

If a data value is produced by an instruction in one processor and consumed by one or more instructions in the other processor, then that data value needs to be communicated to the other processor. A consequence of exceeding the imposed bandwidth limitation is the addition of idle computation cycles while an instruction waits for its required operand to be transferred. In our model, we assume opportunistic use of bandwidth, allowing communication of a value as soon as it is ready, in parallel with computation.

Each data value to be transferred is sent sequentially and occupies the communication channel for a specific amount of time. Data values can be sent any time after the instruction producing the value executes, but must arrive before the first instruction that consumes the value is executed. Data transfers are scheduled onto the communication channel using an "earliest deadline first" algorithm, which produces a scheduling with a minimum of added idle cycles.

Bandwidth constraints are applied by changing the amount of time each data value occupies on the communication channel. Communication latency is applied by setting the deadline for a value some number of cycles after the value is produced.

Computing the bandwidth requirements and idle cycles needed, and thus the cost to switch modes, requires a scheduling of the instruction trace, but the optimal instruction trace scheduling is affected by the cost of switching modes. We approximate the ideal behavior by iteratively performing scheduling using a constant mode switch overhead for each mode switch and then updating the average penalty due to bandwidth consumption across all mode switches, then using the new estimate of average switch cost as input into the scheduling algorithm, until convergence.

3 Simulation Infrastructure

We evaluate performance using micro-op traces extracted from execution of a set of x86-64 benchmarks on the PTLsim [18] simulator. Each micro-op trace was then scheduled using our scheduling algorithm for execution on the heterogeneous system.

3.1 Benchmark Set

We chose our benchmarks with a focus towards general-purpose computing. We used the reference workloads for SPECint and SPECfp 2000 v1.3.1 (23 benchmarks, except 253.perlbmk and 255.vortex which did not run in our simulation environment), PhysicsBench 2.0 [31] (8 benchmarks), SimpleScalar 3.0 [29] (used here as a benchmark), and four small microbenchmarks (described in Table 1).

Benchmark	*Description*
`linear`	Compute average of 9 input pixels for each output pixel. Each pixel is independent.
`sepia`	3×3 constant matrix multiply on each pixel's 3 components. Each pixel is independent.
`serial`	A long chain of dependent instructions, has parallelism approximately 1 (no parallelism).
`twophase`	Loops through two alternating phases, one with no parallelism, one with high parallelism. Needs to switch between processor types for high speedup.

Table 1. Microbenchmarks

We chose PhysicsBench because it contains both sequential and parallel phases in the benchmark, and would be a likely candidate to benefit from heterogeneity, as it would be unsatisfactory if both types of phases were constrained to one processor type [31].

Our SimpleScalar benchmark used the out-of-order processor simulator from SimpleScalar/PISA, running `go` from SPECint 95, compiled for PISA.

We used four microbenchmarks to observe behavior at extremes of parallelism, as shown in Table 1. `Linear` and `sepia` are highly parallel, `serial` is serial, and `twophase` has alternating highly parallel and serial phases.

Figure 2 shows the average parallelism present in our benchmark set. As expected, SPECfp has more parallelism (611) than SPECint (116) and PhysicsBench (83). `Linear` (4790) and `sepia` (6815) have the highest parallelism, while `serial` has essentially no parallelism.

3.2 Traces

Micro-op traces were collected from PTLsim running x86-64 benchmarks, compiled with gcc 4.1.2 -O2. Four microbenchmarks were run in their entirety, while the 32 real benchmarks were run through SimPoint [30] to choose representative sub-traces to analyze. Our traces are captured at the micro-op level, so in this paper instruction and micro-op are used interchangeably.

We used SimPoint to select simulation points of 10-million micro-ops in length from complete runs of benchmarks. As recommended [30], we allowed Sim-

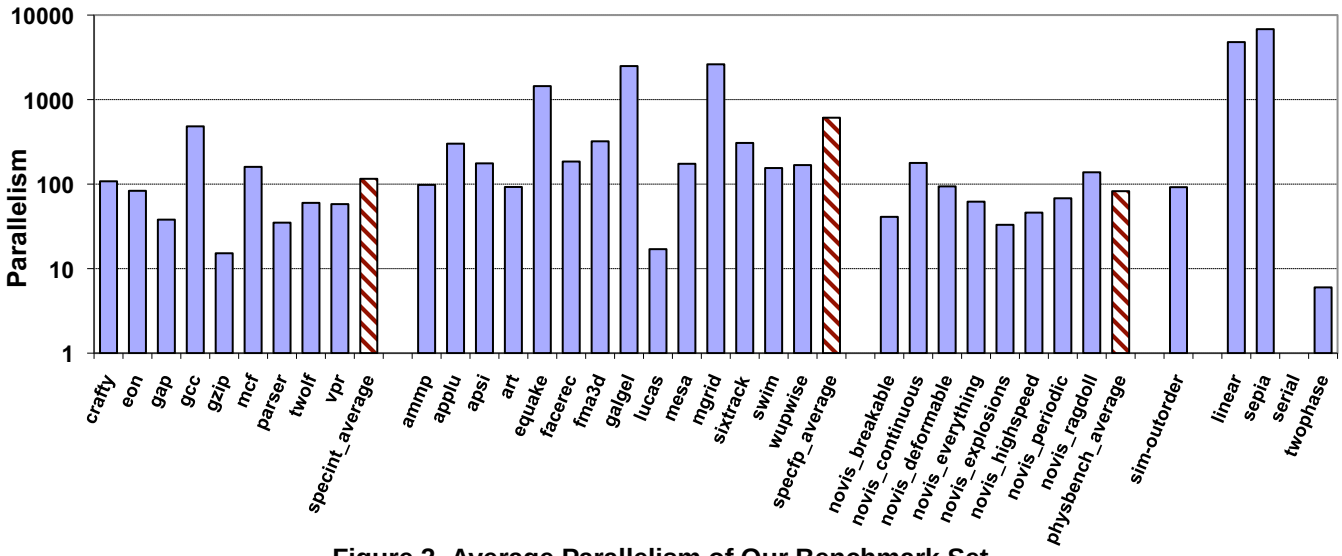

Figure 2. Average Parallelism of Our Benchmark Set

Point to decide how many simulation points should be used to approximate the entire benchmark run. We averaged 12.9 simulation points per benchmark. This is a significant savings over the complete benchmarks which were typically several hundred billion instructions long. The weighted average of the results over each set of SimPoint traces are presented for each benchmark.

We assume branches are correctly predicted. Many branches, like loops, can often be easily predicted or speculated or even restructured away during manual parallelization. As we are trying to evaluate the upper-bound of parallelism in an algorithm, we avoid limiting parallelism by not imposing the branch-handling characteristics of sequential machines. This is somewhat optimistic as true data-dependent branches would at least need be converted into speculation or predicated instructions.

Each trace is analyzed for true data dependencies. Register dependencies are recognized if an instruction consumes a value produced by an earlier instruction (read-after-write). Dependencies on values carried by the instruction pointer register are ignored, to avoid dependencies due to instruction-pointer-relative data addressing. Like earlier limit studies [17, 15], stack pointer register manipulations are ignored, to extract parallelism across function calls. Memory disambiguation is perfect: Dependencies are carried through memory only if an instruction loads a value from memory actually written by an earlier instruction.

It is also important to be able to extract loop-level parallelism and avoid serialization of loops through the loop induction variable. We implemented a generic solution to prevent this type of serialization. We identify instructions that produce result values that are statically known, which are instructions that have no input operands (e.g. load constant). We then repeatedly look for instructions dependent only on values that are statically known and mark the values they produce as statically known as well. We then remove dependencies on all statically-known values. This is similar to repeatedly applying constant folding and constant propagation optimizations [20] to the instruction trace. The dead code that results is removed as described in Section 2.3.

A loop induction variable [20] is often initialized with a constant (e.g. 0). Incrementing the induction variable by a constant depends only on the initialization value of the induction variable, so the incremented value is also statically known. Each subsequent increment is likewise statically known. This removes serialization caused by the loop control variable, but preserves genuine data dependencies between loop iterations, including loop induction variable updates that depend on a variable computed value.

4 Results

In this section, we present our analysis of our experimental results. First, we look at the speedup that can be achieved when adding a parallel co-processor to a sequential machine and show that the speedup is highly dependent on the parallel instruction latency. We define *parallel instruction latency* as the ratio of the read-after-write latency of the parallel cores (recall we assume a CPI of one for the sequential core). We then look at the effect of communication latency and bandwidth as parallel instruction latency is varied, and see that the effect is significant, but small.

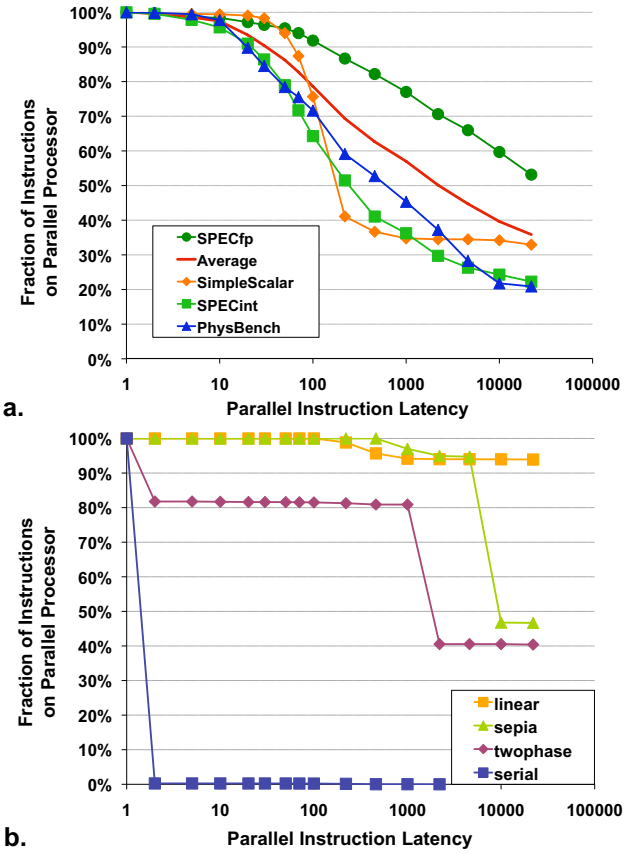

Figure 3. Proportion of Instructions Scheduled on Parallel Core. Real benchmarks (a), Microbenchmarks(b)

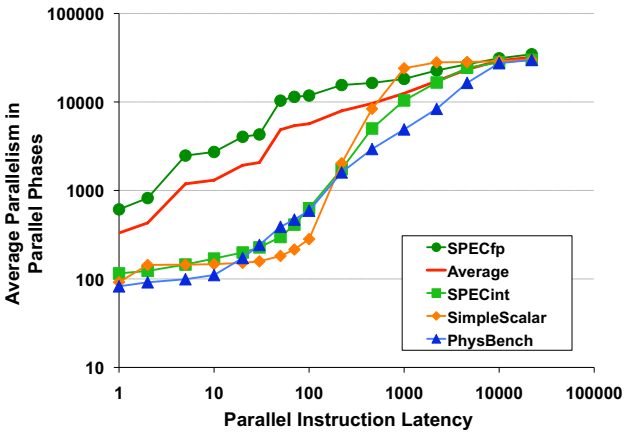

Figure 4. Parallelism on Parallel Processor

4.1 Why Heterogeneous?

Figures 3 and 4 give some intuition for the characteristics of the scheduling algorithm. Figure 4 shows the parallelism of the instructions that are scheduled to use the parallel processor when our workloads are scheduled for best performance. Figure 3(a) shows the proportion of instructions that are assigned to execute on the parallel processor. As the instruction latency increases, sections of the workload where the benefit of parallelism does not outweigh the cost of slower sequential performance become scheduled onto the sequential processor, raising the average parallelism of those portions that remain on the parallel processor, while reducing the proportion of instructions that are scheduled on the parallel processor. The instructions that are scheduled to run on the sequential processor receive no speedup, but scheduling more instructions on the parallel processor in an attempt to increase parallelism will only decrease speedup.

The microbenchmarks in Figure 3(b) show our scheduling algorithm works as expected. `Serial` has nearly no instructions scheduled for the parallel core. `Twophase` has about 18.5% of instructions in its serial component that are scheduled on the sequential processor leaving 81.5% on the parallel processor, while `sepia` and `linear` highly prefer the parallel processor.

We look at the potential speedup of adding a parallel processor to an existing sequential machine. Figures 5(a) and (b) show the speedup of our benchmarks for varying parallel instruction latency, as a speedup over a single sequential processor. Two plots for each benchmark group are shown: The solid plots show the speedup of a heterogeneous system where communication has no cost, while the dashed plot shows speedup when communication is very expensive. We focus on the solid plots in this section.

It can be observed from Figures 5(a) and (b) that as the instruction latency increases, there is a significant loss in the potential speedup provided by the extra parallel processor, becoming limited by the amount of parallelism available in the workload that can be extracted, as seen in Figure 3. Since our parallel processor model is somewhat optimistic, the speedups shown here should be regarded as an upper bound of what can be achieved.

With a parallel processor with GPU-like instruction latency of 100 cycles, SPECint would be limited to a speedup of 2.2×, SPECfp to 12.7×, PhysicsBench to 2.5×, with 64%, 92%, and 72% of instructions scheduled on the parallel processor, respectively. The speedup is much lower than the peak relative throughput of a GPU compared to a sequential CPU ($\approx 50\times$), which shows that if a GPU-like processor were used as the parallel processor in a heterogeneous system, the speedup on these workloads would be limited by the parallelism available in the workload, while still leaving much of the GPU hardware idle.

In contrast, for highly-parallel workloads, the speedups achieved at an instruction latency of 100 are

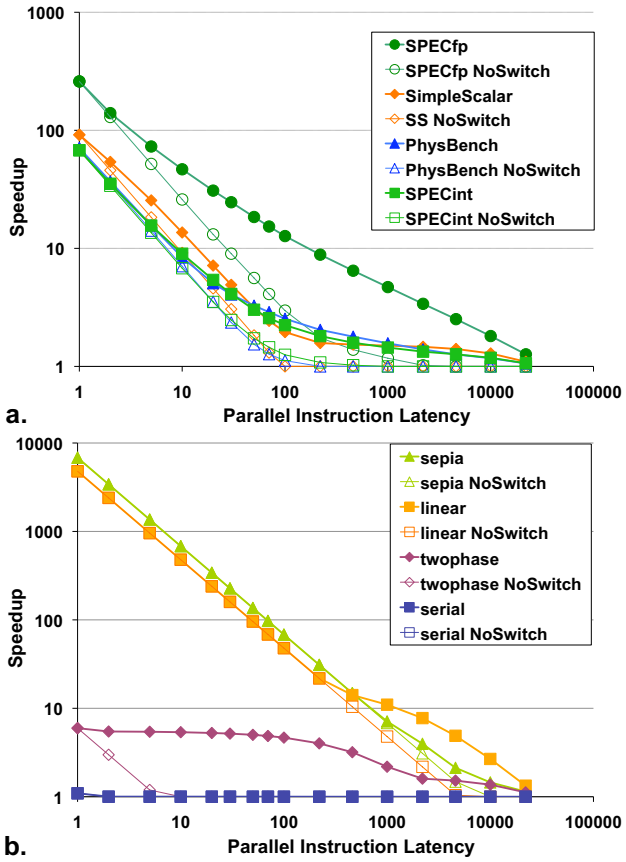

Figure 5. Speedup of Heterogeneous System: (a) Real benchmarks, (b) Microbenchmarks. Ideal communication (solid), communication forbidden (dashed, NoSwitch).

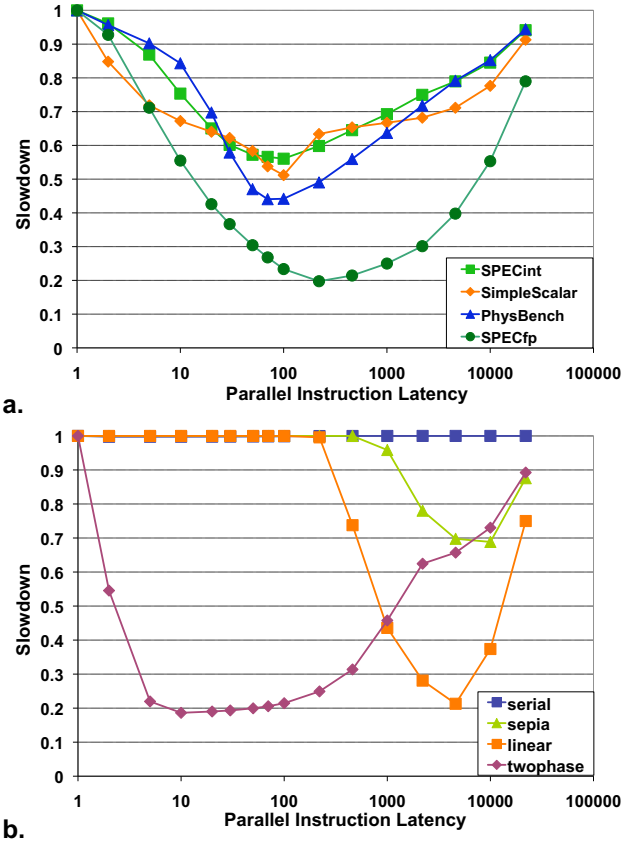

Figure 6. Slowdown of infinite communication cost (NoSwitch) compared to zero communication cost. Real benchmarks (a), Microbenchmarks (b).

similar to the peak throughput available in a GPU. The highly-parallel linear filter and sepia tone filter (Figure 5(b)) kernels have enough parallelism to achieve 50-70× speedup at an instruction latency of 100. A highly-serial workload (serial) does not benefit from the parallel processor.

Although current GPU compute solutions built with efficient low-complexity multi-threaded cores are sufficient to accelerate algorithms with large amounts of thread-level parallelism, general-purpose algorithms would be unable to utilize the large number of thread contexts provided by the GPU, while under-utilizing the arithmetic hardware available.

4.2 Communication

In this section, we evaluate the impact of communication latency and bandwidth on the potential speedup, comparing performance between the extreme cases where communication is unrestricted and communication is forbidden. The solid plots in Figure 5 show speedup when there are no limitations on communication, while the dashed plots (marked NoSwitch) has communication so expensive that the scheduler chooses to run the workload entirely on the sequential processor or parallel processor, never switching between them. Figures 6(a) and (b) show the ratio between the solid and dashed plots in Figures 5(a) and (b), respectively, to highlight the impact of communication. At both extremes of instruction latency, where the workload is mostly sequential or mostly parallel, communication has little impact. It is in the moderate range around 100-200 where communication potentially matters most.

The potential impact of expensive (latency and bandwidth) communication is significant. For example, at a GPU-like instruction latency of 100, SPECint achieves only 56%, SPECfp 23%, and PhysicsBench 44% of the performance of no communication, as can be seen in Figure 6(a). From our microbenchmark set (Figures 5(b) and 6(b)), `twophase` is particularly sensitive to communication costs, and gets no speedup for instruction latency above 10. We look at more realistic

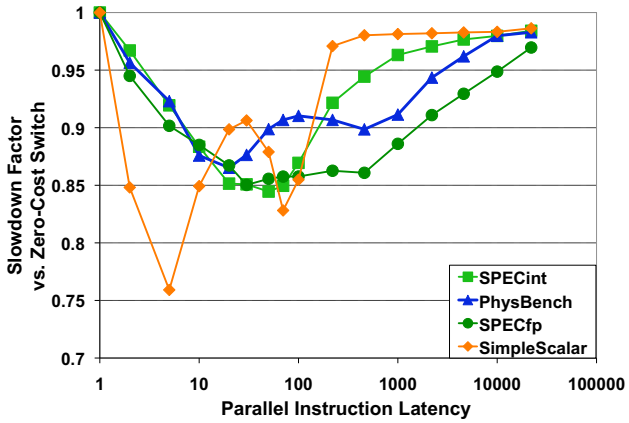

Figure 7. Slowdown due to 100,000 cycles of mode-switch latency. Real benchmarks.

constraints on latency and bandwidth in the following sections.

4.2.1 Latency

Poor parallel performance is often attributed to high communication latency [21]. Heterogeneous processing adds a new communication requirement—the communication channel between sequential and parallel processors (Figure 1). In this section, we measure the impact of the latency of this communication channel.

We model this latency by requiring that switching modes between the two processor types causes a fixed amount of idle computation time. In this section, we do not consider the bandwidth of the data that needs to be transferred. This model represents a heterogeneous system with shared memory (Figure 1(a)), where migrating a task does not involve data copying, but only involves a pipeline flush, notification to the other processor of work, and potentially flushing private caches if caches are not coherent.

Figure 7 shows the slowdown when we include 100,000 cycles of mode-switch latency in our performance model and scheduling, when compared to zero-latency mode switch.

The impact of imposing a delay for every mode switch has only a minor effect on runtime. Although Figure 6(a) suggested that the potential for performance loss due to latency is great, even when each mode switch costs 100,000 cycles (greater than 10us at current clock rates), most of the speedup remains. We can achieve ≈85% of the performance of a heterogeneous system with zero-cost communication. Stated another way, reducing latency between sequential and parallel cores might provide an average ≈ 18% performance improvement.

To gain further insight into the impact of mode

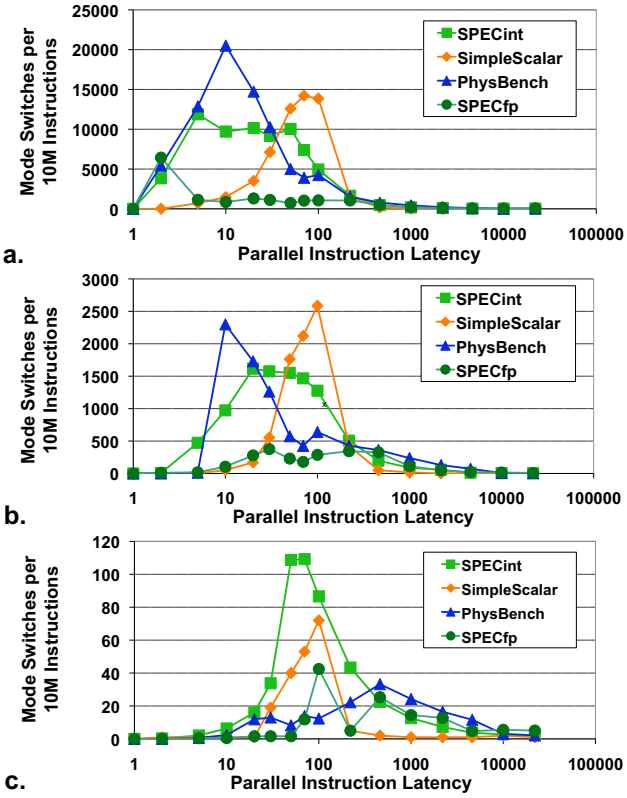

Figure 8. Mode switches as switch latency varies: (a) zero cycles, (b) 10 cycles, (c) 1000 cycles.

switch latency, Figure 8 illustrates the number of mode switches per 10 million instructions as we vary the cost of switching from zero to 1000 cycles. As the cost of a mode switch increases the number of mode switches decreases. Also, more mode switches occur at intermediate values of parallel instruction latency where the benefit of being able to use both types processors outweighs the cost of switching modes.

For systems with private memory (e.g. discrete GPU), data copying is required when migrating a task between processors at mode switches. We consider bandwidth constraints in the next section.

4.2.2 Bandwidth

In the previous section, we saw that high communication latency had only a minor effect on achievable performance. Here, we place a bandwidth constraint on the communication between processors. Data that needs to be communicated between processors is restricted to a maximum rate, and the processors are forced to wait if data is not available in time for an instruction to use it, as described in Section 2.3.3. We also include 1,000 cycles of latency as part of the model.

We first construct a model to represent PCI Express,

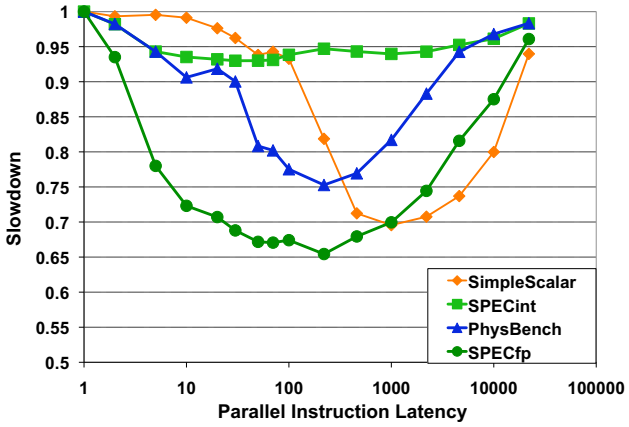

Figure 9. Slowdown due to a bandwidth constraint of 8 cycles per 32-bit value and 1,000 cycles latency, similar to PCI Express x16. Real benchmarks.

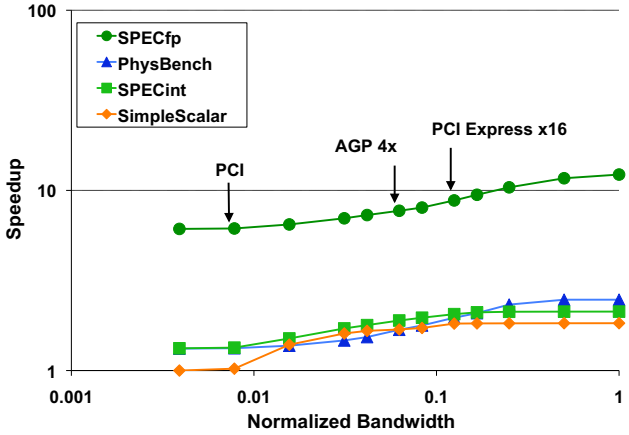

Figure 10. Speedup over sequential processor for varying bandwidth constraints. Real benchmarks.

as discrete GPUs are often attached to the system this way. PCI Express x16 has a peak bandwidth of 4GB/s and latency around 250ns [4]. Assuming current processors perform about 4 billion instructions per second on 32-bit data values, we can model PCI Express using a latency of about 1,000 cycles and bandwidth of 4 cycles per 32-bit value. Being somewhat pessimistic to account for overheads, we use a bandwidth of 8 cycles per 32-bit value (about 2GB/s).

Figure 9 shows the performance impact of restricting bandwidth to one 32-bit value every 8 clocks with 1,000 cycles of latency. Slowdown is worse than with 100,000 cycles of latency, but the benchmark set affected the most (SPECfp) can still achieve ≈67.4% of the ideal performance at a parallel instruction latency of 100. Stated another way, increasing bandwidth between sequential and parallel cores might provide an average 1.48× performance improvement for workloads like SPECfp. For workloads such as PhysicsBench and SPECint the potential benefits appear lower (1.33× and 1.07× potential speedup, respectively). Comparing latency (Figure 7) to bandwidth (Figure 9) constraints, SPECfp and PhysicsBench has more performance degradation than under a pure-latency constraint, but SPECint performs better, suggesting that SPECint is less sensitive to bandwidth.

The above plots suggest that a heterogeneous system attached without a potentially-expensive, low-latency, high-bandwidth communication channel can still achieve much of the potential speedup.

To further evaluate whether GPU-like systems could be usefully attached using even lower bandwidth interconnect, we measure the sensitivity of performance to bandwidth for instruction latency 100. Figure 10 shows the speedup for varying bandwidth. Bandwidth (x-axis) is normalized to 1 cycle per datum, equivalent to about 16GB/s in today's systems. Speedup (y-axis) is relative to the workload running on a sequential processor.

SPECfp and PhysicsBench have similar sensitivity to reduced bandwidth, while SPECint's speedup loss at low bandwidth is less significant (Figure 10). Although there is some loss of performance at PCI Express speeds (normalized bandwidth = 1/8), about half of the potential benefit of heterogeneity remains at PCI-like speeds (normalized bandwidth = 1/128). At PCI Express x16 speeds, SPECint can achieve 92%, SPECfp 69%, and PhysicsBench 78% of the speedup achievable without latency and bandwidth limitations.

As can be seen from the above data, heterogeneous systems can potentially provide significant performance improvements on a wide range of applications, even when system cost sensitivity demands high-latency, low-bandwidth interconnect. However, it also shows that applications are not entirely insensitive to latency and bandwidth, so high-performance systems will still need to worry about increasing bandwidth and lowering latency.

The lower sensitivity to latency than to bandwidth suggests that a shared-memory multicore heterogeneous system would be of benefit, as sharing a single memory system avoids data copying when migrating tasks between processors, leaving only synchronization latency. This could increase costs, as die size would increase, and the memory system would then need to support the needs of both sequential and parallel processors. A high-performance off-chip interconnect like PCI Express or HyperTransport may be a good compromise.

5 Related Work

There have been many limit studies on the amount of parallelism within sequential programs.

Wall [7] studies parallelism in SPEC92 under various limitations in branch prediction, register renaming, and memory disambiguation. Lam et al. [17] studies parallelism under branch prediction, condition dependence analysis, and multiple-fetch. Postiff et al. [15] perform a similar analysis on the SPEC95 suite of benchmarks. These studies showed that significant amounts of parallelism exist in typical applications under optimistic assumptions. These studies focused on extracting instruction-level parallelism on a single processor. As it becomes increasingly difficult to extract ILP out of a single processor, performance increases often comes from multicore systems.

As we move towards multicore systems, there are new constraints, such as communication latency, that are now applicable. Vachharajani et al. [21] studies speedup available on homogeneous multiprocessor systems. They use a greedy scheduling algorithm to assign instructions to cores. They also scale communication latency between cores in the array of cores and find that it is a significant limit on available parallelism.

In our study, we extend these analyses to heterogeneous systems, where there are two types of processors. Vachharajani examined the impact of communication between processors within a homogeneous processor array. We examine the impact of communication between a sequential processor and an array of cores. In our model, we roughly account for communication latency between cores within an array of cores by using higher instruction read-after-write latency.

Heterogeneous systems are interesting because they are commercially available [10, 25, 2] and, for GPU compute systems, can leverage the existing software ecosystem by using the traditional CPU as its sequential processor. They have also been shown to be more area and power efficient [16, 26, 27] than homogeneous multicore systems.

Hill and Marty [16] uses Amdahl's Law to show that there are limits to parallel speedup, and makes the case that when one must trade per-core performance for more cores, heterogeneous multiprocessor systems perform better than homogeneous ones because non-parallelizable fragments of code do not benefit from more cores, but do suffer when all cores are made slower to accommodate more cores. They indicate that more research should be done to explore "the scheduling and overhead challenges that Amdahl's model doesn't capture". Our work can be viewed as an attempt to further quantify the impact that these challenges present.

6 Conclusion

We conducted a limit study to analyze the behavior of a set of general purpose applications on a heterogeneous system consisting of a sequential processor and a parallel processor with higher instruction latency.

We showed that instruction read-after-write latency of the *parallel* processor was a significant factor in performance. In order to be useful for applications without copious amounts of parallelism, we believe that instruction read-after-write latencies of GPUs will need to decrease and thus GPUs can no longer rely exclusively on fine-grain multithreading to keep utilization high. We note that VLIW or superscalar issue combined with fine-grained multithreading [3, 19] do not inherently mitigate this read-after-write latency, though adding forwarding [12] might. Our data shows that latency and bandwidth of communication between the parallel cores and the sequential core, while significant factors, have comparatively minor effects on performance. Latency and bandwidth characteristics of PCI Express was sufficient to achieve most of the available performance.

Note that since our results are normalized to the sequential processor, our results scale as processor designs improve. As sequential processor performance improves in the future, the read-after-write latency of the parallel processor will also need to improve to match.

Manufacturers have and will likely continue to build single-chip heterogeneous multicore processors. The data presented in this paper may suggest the reasons for doing so are other than to obtain higher performance from reduced communication overheads on general purpose workloads. A subject for future work is evaluating whether such conclusions hold under more realistic evaluation scenarios (limited hardware parallelism, detailed simulations, real hardware) along with exploration of a wider set of applications (ideally including real workloads carefully tuned specifically for a tightly coupled single-chip heterogeneous system). As well, this work does not quantify the effect that increasing problem size [11] may have on the question of the benefits of heterogeneous (or asymmetric) multicore performance.

Acknowledgements

We thank Dean Tullsen, Bob Dreyer, Hong Wang, Wilson Fung and the anonymous reviewers for their comments on this work. This work was partly supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] AMD Inc. The future is fusion. http://sites.amd.com/us/Documents/AMD_fusion_Whitepaper.pdf, 2008.

[2] AMD Inc. *ATI Stream Computing User Guide*, 2009.

[3] AMD Inc. *R700-Family Instruction Set Architecture*, 2009.

[4] B. Holden. Latency Comparison Between HyperTransport and PCI-Express in Communications Systems. http://www.hypertransport.org/.

[5] J. A. Butts and G. Sohi. Dynamic Dead-Instruction Detection and Elimination. In *Proc. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 199–210, 2002.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw Hill, 2nd edition, 2001.

[7] D. W. Wall. Limits of Instruction-Level Parallelism. Technical Report 93/6, DEC WRL, 1993.

[8] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware. *To appear in: ACM Trans. Architec. Code Optim. (TACO)*, 2009.

[9] G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conf. Proc. vol. 30*, pages 483–485, 1967.

[10] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, D. Shippy. Introduction to the Cell multiprocessor. *IBM J. R&D*, 49(4/5):589–604, 2005.

[11] J. L. Gustafson. Reevaluating Amdahl's Law. *Communications of ACM*, 31(5):532–533, 1988.

[12] J. Laudon, A. Gupta, and M. Horowitz. Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations. In *ASPLOS*, pages 308–318, 1994.

[13] E. Lindholm, M. J. Kligard, and H. P. Moreton. A user-programmable vertex engine. In *SIGGRAPH*, pages 149–158, 2001.

[14] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, March-April 2008.

[15] M. A. Postiff, D. A. Greene, G. S. Tyson, T. N. Mudge. The Limits of Instruction Level Parallelism in SPEC95 Applications. *ACM SIGARCH Comp. Arch. News*, 27(1):31–34, 1999.

[16] M. D. Hill, M. R. Marty. Amdahlś Law in the Multicore Era. *IEEE Computer*, 41(7):33–38, 2008.

[17] M. S. Lam, R. P. Wilson. Limits of control flow on parallelism. In *Proc. 19th Int'l Symp. on Computer Architecture*, pages 46–57, 1992.

[18] M. T. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *IEEE Int'l Symp. on Performance Analysis of Systems and Software ISPASS*, 2007.

[19] J. Montrym and H. Moreton. The GeForce 6800. *IEEE Micro*, 25(2):41–51, 2005.

[20] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kauffman, 1997.

[21] N. Vachharajani, M. Iyer, C. Ashok, M. Vachharajani, D. I. August, D. Connors. Chip multi-processor scalability for single-threaded applications. *ACM SIGARCH Comp. Arch. News*, 33(4):44–53, 2005.

[22] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[23] NVIDIA Corp. GeForce 8 Series. http://www.nvidia.com/page/geforce8.html.

[24] NVIDIA Corp. GeForce GTX 280. http://www.nvidia.com/object/geforce_gtx_280.html.

[25] NVIDIA Corp. *CUDA Programming Guide*, 2.2 edition, 2009.

[26] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proc. 31st Int'l Symp. on Computer Architecture*, page 64, 2004.

[27] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, D. M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proc. 36th IEEE/ACM Int'l Symp. on Microarchitecture*, page 81, 2003.

[28] S. Borkar. Thousand Core Chips — A Technology Perspective. In *Proc. 44th Annual Conf. on Design Automation*, pages 746–749, 2007.

[29] T. Austin, E. Larson, D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2), 2002.

[30] T. Sherwood, E Perelman, G. Hamerly, B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proc. 10th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems ASPLOS*, 2002.

[31] T. Y. Yeh, P. Faloutsos, S. J. Patel, G. Reinman. ParallAX: an architecture for real-time physics. In *Proc. 34th Int'l Symp. on Computer Architecture*, pages 232–243, 2007.

[32] J. E. Thornton. Parallel operation in the control data 6600. In *AFIPS Proc. FJCC*, volume 26, pages 33–40, 1964.

[33] H. Wong, A. Bracy, E. Schuchman, T. M. Aamodt, J. D. Collins, P. H. Wang, G. Chinya, A. K. Groen, H. Jiang, and H. Wang. Pangaea: A Tightly-Coupled IA32 Heterogeneous Chip Multiprocessor. In *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT 2008)*, pages 52–61, Oct. 2008.