Increasing Memory Miss Tolerance for SIMD Cores

David Tarjan,[*] Jiayuan Meng and Kevin Skadron
Department of Computer Science
University of Virginia, Charlottesville, VA 22904
{dtarjan, jm6dg,skadron}@cs.virginia.edu

ABSTRACT

Manycore processors with wide SIMD cores are becoming a popular choice for the next generation of throughput oriented architectures. We introduce a hardware technique called "diverge on miss" that allows SIMD cores to better tolerate memory latency for workloads with non-contiguous memory access patterns. Individual threads within a SIMD "warp" are allowed to slip behind other threads in the same warp, letting the warp continue execution even if a subset of threads are waiting on memory. Diverge on miss can either increase the performance of a given design by up to a factor of 3.14 for a single warp per core, or reduce the number of warps per core needed to sustain a given level of performance from 16 to 2 warps, reducing the area per core by 35%.

Categories and Subject Descriptors

C.1 [**PROCESSOR ARCHITECTURES**]: Multiple Data Stream Architectures (Multiprocessors)

1. INTRODUCTION

The growth in single-thread performance has slowed dramatically in recent years, due to limits in the power consumption, thermal constraints and complexity. As a response, the microprocessor industry has shifted its focus onto multicore processors, which combine a number of cores onto a single die. Some of these designs give higher priority to overall throughput than to single-thread latency, trading out-of-order cores for simpler, smaller in-order cores which are smaller and less power hungry. While single-thread performance suffers, overall chip throughput is increased. This design point is often referred to as "manycore", as opposed to more traditional multicore designs, which retain an organization with few high-performance out-of-order cores for maximum single-thread performance.

Manycore designs are limited by both the overall power budget and the available chip area. Manycore processors that use a *single instruction multiple data* (SIMD) organization can amortize the area and power overhead of a single frontend over a large number of execution backends. For example, we estimate that a 32-wide SIMD core requires about one fifth the area of 32 individual scalar cores. Note that this estimate does not include the area of any interconnection network among the MIMD cores, which often grows supra-linearly with the number of cores [18].

To better tolerate memory and pipeline latencies, manycore processors typically use fine-grained multi-threading, switching among multiple warps,[1] so that active warps can mask stalls in other warps waiting on long-latency events. The drawback of this approach is that the size of the register file increases along with the number of warps per core. Most current and planned manycore processors also use on-chip caches to reduce the required off-chip bandwidth and to hide the latency of accessing DRAM as much as possible. The combination of SIMD cores and caches presents special problems for architects because each SIMD thread may independently hit or miss. This problem is not just limited to *array-style* SIMD organizations where each SIMD thread is a scalar processing element. *Vector-SIMD* instructions sets with gather support, including [1, 25] suffer the same problem. Divergence becomes a particular problem for load or store instructions that have irregular access patterns. Consider code where each thread of a SIMD warp needs to read many contiguous values in a global array, but each thread accesses distinct regions, starting at a random offset, for example in DNA sequence alignment. While reading in their values, the probability that a thread in a warp will cross a cache line boundary and have to stall grows as the number of threads per warp increases. In such a case, the lockstep nature of SIMD execution forces the core to stall or switch to another warp for each load. Clearly, such memory access patterns will waste much of the computational power of the SIMD core waiting on memory requests.

This paper presents a new hardware mechanism, *diverge on miss*, that takes advantage of looping behavior to temporarily mask off threads in a warp that miss in the data cache and allows the other threads to continue executing, re-enabling the masked off threads as soon as possible. Letting threads which hit in the cache continue to execute allows them to use idle execution slots when all warps of a core would otherwise be stalled. It also allows them to issue

[*]Now with NVIDIA Research

[1]For simplicity, we use the term *thread* to refer to a SIMD lane, and *warp* to a SIMD group that operates in lockstep. Multithreading a SIMD core therefore consists of supporting multiple warps.

future cache misses earlier, increasing memory level parallelism [16]. We call warps where some threads have continued to execute while others are stalled waiting on memory *slipping warps*.

We show that diverge on miss can increase performance of a manycore processor using 32-wide SIMD cores by up to a factor of 3.14 over an architecture which doesn't incorporate diverge on miss, can decrease the area of each SIMD core by 35% at equal performance or increase peak performance by 30% compared to an architecture which doesn't use diverge miss. We show how such a mechanism can be built with low overhead on top of existing structures meant to deal with control-flow divergence. Diverge on miss builds on the fact that high-performance SIMD and vector cores *already have* logic for masking off threads on a fine-grained basis to support arbitrary control-flow and can already deal with multiple parallel memory operations finishing out-of-order due to their support of scatter/gather operations.

2. RELATED WORK

Early academic work [7, 22] on manycore processors explored the benefit of chips built out of many simple cores for both commercial and scientific workloads. They showed that for workloads with sufficient parallelism, many simple cores could outperform an organization with few high-performance cores. Recent commercial, general-purpose products that target throughput-oriented workloads exemplify some of these lessons. For example, the Niagara processor [2] from Sun implements 8 simple SPARC cores, each of which has 4 execution contexts.

GPU manufacturers have evolved their designs from pure ASICs to general-purpose manycore processors, with each core having a logical warp width between 32 and 64 and a large number of warps per core [4, 17]. While all of this hardware was traditionally hidden behind complex graphics APIs, recently both AMD and NVIDIA have made available APIs [3, 10, 15] which are meant for general purpose computation and can take advantage of GPU hardware.

The recently announced Intel Larrabee architecture [25] has capabilities of both GPUs and multicore processors, supporting both the x86 ISA, cache coherence and memory ordering, as well as wide SIMD execution and multiple hardware execution contexts per core. Both Niagara and Larrabee (will) support conventional cache architectures, where caches are coherent, addressed through a unified address space, obey a well-defined memory ordering model and large enough to hold the working set of multiple programs.

GPUs on the other hand, because they have been designed primarily to support graphics APIs such as OpenGL and Direct3D [8], have very different cache architectures. One major difference is simply in the size of caches relative to number of ALUs, with GPUs having more ALUs relative to cache size than CPUs. Another difference is that caches are divided among different address spaces (so called texture and constant caches) and optimized for specific access patterns which go along with these address spaces in graphics applications. A variety of CUDA applications have taken advantage of these properties, Che *et al.* [11] and Boyer *et al.* [9] in particular discuss the importance of using these memory paths.

In general it can be said that GPUs have designed their cache architectures to help maximize aggregate throughput, but not necessarily to minimize the latency of any individual thread. Diverge on miss enables the combination of very wide SIMD execution of GPUs with regular cache hierarchies and helps greatly reduce single-thread latency and increase throughput for workloads with irregular access patterns.

Warp divergence in SIMD processors as a result of control-flow divergence was explored by Fung *et al.* [14], who proposed Dynamic Warp Formation as a way to lessen the performance loss due to this particular problem. While the technique of dynamic warp formation can also be applied to memory divergence, the hardware overhead of our technique is much smaller, requiring only small additions to existing structures. For example, Dynamic Warp Formation requires that the register file have as many independent banks as there are threads in a warp, substantially increasing the area overhead due to addresses having to be routed to each bank, each bank needing its own address decoders and also having much shorter word lines. Our technique requires only one bank for the width of the warp.

The software controlled approach to diverge on miss outlined in Section 4.2 can be compared to the streaming approach of the Merrimac architecture [12] and the Cell chip's Synergistic Processing Units [13]. These architectures have explicit memory hierarchies and independent DMA engines, which can fetch lists of memory references into a large software controlled on-chip buffer asynchronously, without having to block execution.

In contrast to these architectures, a software implementation of diverge on miss does not force the programmer to explicitly organize data in a fixed size buffer, nor does it fix the size of this buffer. Any program written for a von Neumann architecture will work on such a processor. The extra instructions to snoop the memory hierarchy only provide potentially higher performance.

There has been considerable work done to allow single-threaded CPUs to continue to execute instructions and find more cache misses in the shadow of an initial long-latency miss. Examples include Continual Flow Pipelines [26], Runahead Execution [19] and Flea-Flicker Pipelining [6]. Such techniques enhance and extend existing structures for speculative execution of a single thread to allow a large number of instructions from the same thread to be in flight at the same time. Our work focuses on exposing more of the MLP inherent in the many threads of a SIMD core.

3. BACKGROUND ON SIMD DIVERGENCE HANDLING

3.1 Control-Flow Divergence

The baseline architecture in this study uses the same post-dominator based reconvergence algorithm presented by Fung *et al.* [14]. Each warp is associated with a branch divergence stack, which tracks control flow for all threads in the warp. Each entry in this stack holds three fields: the active PC field, an active threads bitmask and a reconvergence PC field.

If a divergent branch (where some threads evaluate the branch as taken and some as not-taken) is executed, the top of the stack entry is modified to hold the reconvergence PC along with a bitmask of the currently active threads in the warp. A new entry is pushed on the stack consisting of the fall through PC, a bitmask indicating which threads

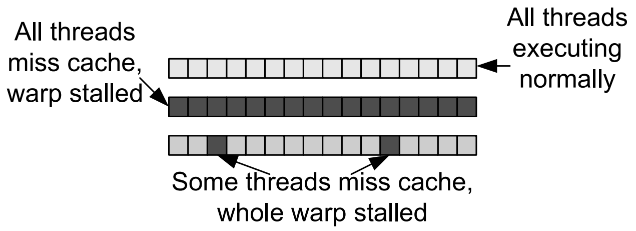
All threads miss cache, warp stalled
All threads executing normally
Some threads miss cache, whole warp stalled

Figure 1: Warps can be forced to wait on memory by a single miss from a single thread. Even cores with multiple warps are forced to stall by a single cache miss per warp.

evaluated the branch as not-taken, as well as the reconvergence PC of the branch. A second entry consisting of the branch-target PC is also pushed on the stack, along with the bitmask indicating which threads evaluated the branch as taken, and again the reconvergence PC.

The active PC and thread-active bitmask are then set to the active PC and bitmask fields of the top of the stack (which is the taken branch entry in this case) and execution continues. When the active PC reaches the reconvergence PC, the stack is popped and the active PC and bitmask are set to the values contained in the not-taken stack entry.

Finally, when the reconvergence PC is reached a second time the active bitmask is restored to it's state before the branch. If a branch is encountered multiple times in a row (such as a loop branch), then no new entry needs to be created on the stack; it is enough to modify the bitmask if any active threads want to exit the loop. As we will show in Section 4, the same basic operations that are needed to support control-flow divergence by the pipeline logic (checking the PC against a PC stored in a structure, taking a predefined action if the PCs match, and modifying the bitmask of active threads based on the result of that action) also support diverge on miss.

3.2 Divergent Scatter/Gather

When a SIMD vector or array core with scatter/gather support and an attached data cache executes a load, the data cache looks up each cache line touched by each load from each thread. If even a single lookup misses, execution of the entire warp has to stall until that miss has been serviced. We call memory operations in which some threads hit and some threads miss *divergent* memory operations.

If a core only has a single warp to execute, it has to stall in such an event. Even a core with multiple warps can be stalled by only a small number of individual memory requests missing the cache, as illustrated in Figure 1.

The cache lines not in the cache on a divergent memory operation are returned by the memory subsystem in arbitrary order. The values requested within those cache lines must now be extracted and written back to the register file. If the architecture allows writing back of individual threads' register values into the SIMD register file as a background operation, no intermediate storage is needed. If this is not the case, a Memory Coalescing Buffer (MCB) is needed, where values are buffered between the time they are read from the cache and when they are written back. An MCB is also needed for those threads which hit in the cache if individual register writes are not allowed. All threads that have hit in the cache must capture their values, as the cache lines they access may be evicted during the servicing of any misses. Each MCB entry must track which threads are waiting on which cache lines, and which threads are active at all, since some threads may be masked off due to control-flow divergence. Each entry thus holds a three state FSM tracking whether a thread is invalid, waiting on memory, or has received its memory value. The MCB also acts as a write buffer for divergent writes, with the write data for each missing thread waiting in the MCB entry for its cache line to be brought into the cache.

4. DIVERGE ON MISS

Diverge on Miss is a hardware mechanism which allows some threads in a warp to continue to execute on divergent memory accesses. Threads which miss in the data cache (or a given cache level if there is a multi-level cache hierarchy) are masked off and do not continue execution, while the threads that hit in the cache continue to execute normally. Such a warp is called a *slipping* warp, as it allows some threads to slip or lag behind others. Figure 2 shows a comparison of execution with normal, blocking SIMD hardware and Diverge on Miss.

Memory requests from missing threads are serviced in parallel with the warp continuing execution. When the warp next encounters the same memory instruction (or a condition which forces re-synchronization of all threads in a warp)[2] the missing threads that have received their memory value in the meantime are re-enabled. Threads which still have not received their memory values continue to be masked off. Individual threads can slip a variable amount relative to other threads, potentially missing the cache shortly after being re-enabled. Slipping warps can either catch up when other threads miss in the cache or continue to execute after the other threads have already finished executing, forcing the warp to execute longer. In general, threads are likely to leapfrog each other, limiting the risk of laggards.

For programs which are memory latency bound, diverge on miss can dynamically trade execution cycles for more latency tolerance, higher MLP and potentially improved utilization of the data cache. We will show in Section 5 how the hardware can use runtime control mechanisms to limit the amount of slip, controlling the amount of extra execution cycles based on the needs of the running program.

We discuss two options for supporting diverge on miss: a pure hardware implementation and a hybrid hardware-software approach which only exposes a new type of load and store instructions, but leaves all the implementation and handling of the divergence to the software layer.

4.1 Pure Hardware Implementation

Diverge on miss uses a very similar structure to the divergence stack used by branch divergence. The Memory Divergence Table (MDT) shown in Figure 3, keeps track of divergent memory operations.

The following actions occur when a divergent memory operation is executed:

1. The memory request is issued to the cache and a bitmask indicating which threads hit and which miss is

[2]Examples of synchronizing conditions include barriers and control-flow instructions such as calls and returns

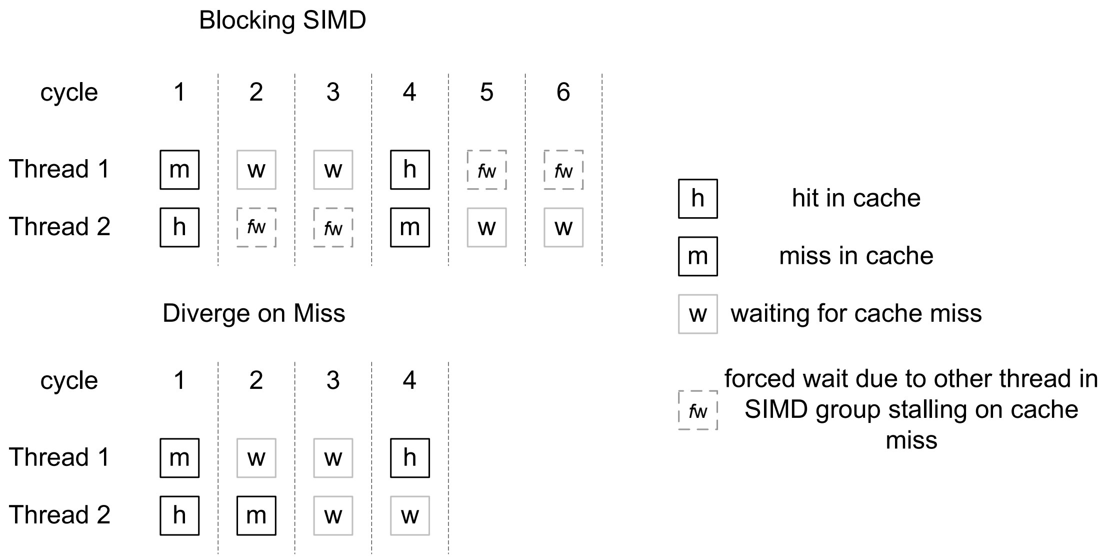

Figure 2: Comparison of a warp with 2 threads executing with Diverge on Miss (bottom) and with normal blocking SIMD execution (top). With blocking SIMD execution, all threads in a warp have to wait if any thread misses in the cache. With Diverge on Miss, a thread which hit in the cache can continue to execute and initiate a subsequent cache miss earlier, allowing a warp to have higher memory level parallelism (MLP) than with blocking SIMD execution.

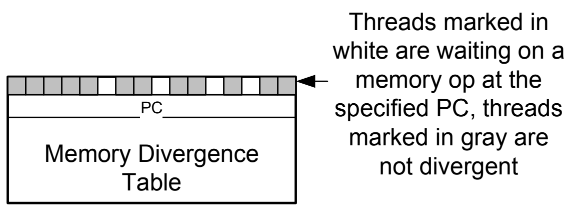
Threads marked in white are waiting on a memory op at the specified PC, threads marked in gray are not divergent
PC
Memory Divergence Table

Figure 3: The Memory Divergence Table tracks which lanes of a warp are waiting on which memory op and which ones are ready to be merged back into the active warp.

returned.

2. The fact that some threads hit and some missed is detected by the control logic.

3. The control logic searches the current warp's MDT entries for an existing entry with the same PC, merging the new request into the MDT entry if it exists. If an MDT entry is not found, the control tries to allocate an MDT entry to the instruction. Allocation might fail because of a limited number of entries per core, or because the adaptive slip controller (described later) decides that it is better to have this memory operation execute as a normal load or store. [3]

   If allocation succeeds, the threads which missed the cache are written to the MDT as a bitmask along with the PC of the memory operation. The MCB entry is also initialized with the memory addresses requested by the threads that missed the cache and the per-thread status fields in the MCB entry are set to either waiting or invalid. The missing addresses are sent to the memory subsystem to be fetched, while threads which hit in the cache receive their memory values and continue executing. If allocation fails, the warp falls-back to normal SIMD execution, and blocks waiting for all misses to complete.

4. Because the time between when a cache line is returned and when a thread can be merged back into the active warp cannot be known a priori, it is possible that a cache line could be evicted while the requesting thread is waiting for re-activation. To prevent this case, as soon as a cache line is returned, the memory values that were requested are extracted and stored in the appropriate slot in the MCB entry. The slots' status bits are also updated from waiting to ready.

5. When the same memory instruction gets executed again (or a forced reconvergence happens), the control logic will again search the MDT and find an existing entry. Threads which have their status bits set to ready will write their fetched memory value back to the register file along with those lanes that hit in the cache, and their status field will be updated to invalid. If all threads have the invalid status, the entry can be deallocated.

4.2 Software-Controlled Implementation

An alternative approach, which we do not evaluate in this paper, is to add a new type of instruction, called the *load&snoop* and *store&snoop*. These instructions operate as normal loads and stores if they hit in the level one data cache (or another level of the cache hierarchy). If they miss,

[3]In either case, an MCB entry is also allocated to the memory operation. If no MCB entry is available, execution has to stall until an entry becomes available.

however, they do not block but are instead turned into implicit prefetches. By guaranteeing a fixed latency to completion, they have the benefit of being easy to schedule for the compiler in optimized loops, similar to accesses to scratchpad memories in other architectures [13].

If a thread misses in the cache, a bit is set in a bitmask. The bitmask can be stored either in a special purpose register or returned as a second register write of the instruction, similar to the low and high parts of a multiplication. In this approach neither the MDT nor the MCB are implemented in hardware. The software can implement most of the functionality of these structures, or modify them according to the needs of the application. Note that because the cache lines which are prefetched are not locked down in any way, a *load&snoop* or *store&snoop* can fail repeatedly and indeed indefinitely. For example if all threads in a warp try to load distinct cache lines that are mapped to a single set in the cache and the cache's associativity is smaller than the width of a warp, it is impossible for all loads to hit in the cache simultaneously. Software can always serialize all loads or stores of a warp if it detects too many retries.

4.3 Ensuring Reconvergence

Supporting SIMD divergence on memory operations raises similar concerns as supporting SIMD branch divergence. Ensuring that all threads get re-merged into the active warp and finish executing requires some extra policies and logic per core.

In typical usage, a divergent load or store will be inside a loop body and executed a large number of times. In this scenario, diverged lanes can normally reconverge on the next iteration of the loop. But if a thread diverges during the last loop iteration or control flow jumps outside the loop body, we must ensure that that thread still reconverges.

If a subset of threads in a warp reach a return statement while other threads are still masked off, the control logic checks the MDT and re-activates those threads while masking off the threads which have hit the return statement. Note that this is the same mechanism that is used to handle branch divergence, so the control logic only has to be extended to check the MDT in addition to the branch divergence stack. If there are multiple entries in the MDT this process is repeated until the MDT is empty.

5. LIMITING THREAD DIVERGENCE

A SIMD core which allows threads to diverge on cache misses has to deal with the potential of excessive divergence. This can occur if some threads hit in the cache the great majority of the time, while the others almost always miss. This can happen due to the inherent nature of a given workload or the interaction of the program with the cache subsystem. In the worst case by the time the fast threads finish executing a loop, the laggard threads will only have advanced a few iterations. The warp containing these threads will have to execute the slow threads to completion, greatly wasting execution cycles and gaining no benefit in terms of overall warp execution latency.

Worse, excessive divergence can make the cache access behavior of a given warp much worse, with accesses that would have been a contiguous, coalesced set of hits turning into accesses spread over many cycles, increasing cache churn and decreasing hit rates. These drawbacks to diverge on miss SIMD execution grow proportionally to the divergence between threads in a warp.

5.1 Adaptive Slip Control

To limit the amount of divergence, we introduce new control hardware, which we call the Adaptive Slip Controller (ASC), to limit how far threads in a warp can slip relative to each other. The ASC has a small counter for each thread in a warp. If an undiverged warp encounters a diverge on miss event, those threads which hit in the cache have their counters incremented. If any thread's counter hits some maximum value, the warp reverts to blocking execution of all loads and stores until the maximum counter value falls below the maximum value again. Note that threads which are marked as inactive by the branch divergence stack are not considered in this process.

If a warp is already diverged when it encounters another diverge event and all tail-end threads (which have counter values of zero) hit in the cache, the counter values of all threads which miss the cache in this instance are decremented. The same mechanism applies when some threads reach the maximum counter value. They are disabled and their counters get decremented when the remaining threads hit in the cache. The counter of each thread is reset when hardware warps are reassigned to a new set of software threads.

5.2 Adaptively Limiting Thread Divergence

The optimal maximum divergence value is very much dependent on the interaction of the program, the input and the architecture. We use a mechanism - called adaptive diff - which keeps track of the number of cycles a core was not actively executing instructions (a value of zero indicating that it is completely ALU bound), whether the amount of off-chip bandwidth that it used was above its fair fraction of overall bandwidth (bandwidth bound), as well as the number of cycles it was stalled waiting on memory (latency bound). Since enabling more slip may result in extra execution cycles (as the trailing threads finish execution) and can result in more bandwidth usage (due to previously coalesced accesses being broken into chunks which are touched at different points in time), the amount of slip is controlled by how ALU-, bandwidth- or latency-bound a given program is during a sampling period. We use very long sampling periods of 100000 cycles or more. If the core was neither ALU nor bandwidth bound over a given sampling period, the maximum allowed divergence value is incremented, if it is either ALU or bandwidth bound the maximum divergence is decremented and kept constant if latency and ALU or bandwidth are roughly in balance.

6. HARDWARE OVERHEAD

Diverge on miss adds the Memory Divergence Table, the per-thread divergence counters and some other small structure to each core. Table 1 lists the extra state required for each structure. The MDT is similar to the branch divergence stack, in that each entry needs to record the PC of a divergent instruction, along with a bitmask indicating which threads took which of the two possible paths. The number of MDT entries per warp is directly related to the maximum number of outstanding memory operations each warp supports. We assume that the baseline architectures allows two outstanding memory operations per warp, which means that the augmented core with diverge on miss has two MDT and

core type	Area
scalar core	1.05 mm^2
32 scalar core	33.60 mm^2
32-wide SIMD core with 2 warps	7.3 mm^2
32-wide SIMD core with 16 warps	11.5 mm^2

Table 2: Area estimates for different core configurations

MCB entries per warp.

As explained in Section 5, it is useful to dynamically adapt the maximum amount of slip allowed among threads in a single warp at runtime. To track the slip of each thread, we need a small counter per thread. We assume that each counter is 8 bits, allowing threads to slip by 255 hits or misses relative to each other. These counters are updated with each divergent memory operation and checked against the Max Slip Counter. If any thread reaches the maximum allowed slip, a bit is set in the warp's Slip-Limit Bitmask, disabling further execution of that thread until divergence is reduced below the threshold value.

6.1 Core Areas

To estimate the areas of SIMD cores versus scalar cores and the impact of different numbers of warps per core we developed an area model using public data and simple scaling rules. To estimate realistic sizes for the different units of a core, we measured the sizes of the different functional units of an AMD Opteron processor in 130nm technology from a publicly available die photo. We could only account for about 70% of the total area, the rest being x86-specific, system level circuits, or unidentifiable. We scaled the functional unit areas to 45nm, assuming a 0.7 scaling factor per generation. Since we assume that each SIMD lane only has a 32 bit data path (combining adjacent lanes if 64 bit results are needed) we scaled all unit areas appropriately.

We use the numbers for each functional unit and scale them by their capacities and port numbers relative to the Opteron core. Table 2 shows the areas for a 32-wide SIMD core with 2 warps, a core with 16 warps, a scalar core and 32 scalar cores calculated with this methodology.

7. EXPERIMENTAL SETUP

7.1 Simulator

Our custom simulator models a set of SIMD/vector cores, along with a cache hierarchy and a shared memory subsystem. The cores are modeled as having a constant CPI of one for all non-memory instructions and private L1 data caches. We assume that the structures for holding outstanding memory requests are not a limiting factor. Each core can have one or multiple warps, and it can switch among them on a cycle by cycle basis at no extra cost. The scheduling algorithm is round-robin, skipping warps which are waiting on memory requests. The memory reference traces are collected directly from the native applications, which are instrumented with calls to our simulator. To determine the number of instructions between memory references, each application is inspected manually and the number of arithmetic and control flow instructions between memory references are passed to the simulator.

Direct instrumentation of native applications was preferred over gathering large memory traces, to avoid the I/O and decompression overheads of normal trace-based simulators. The combination of a simple core model and direct instrumentation of native applications allows the simulator to be very fast (slowdowns of only 10x over pure native execution are the norm) and can consequently model input sizes which would be prohibitively slow to simulate otherwise. This is especially important when dealing with a large number of cores and threads per core.

7.2 Simulated System

Our base chip consists of 32 in-order cores each supporting 32-wide SIMD execution, all running at 2 GHz, for an overall maximum execution bandwidth of 2 Teraops. Each core has a 32KB private data cache, which has 32B cache lines and is 4-way set associative. We model a standard LRU replacement policy. All cores share a 256 GB/sec memory interface, with a memory access latency of 500 cycles.

7.3 Workload

Our chosen application kernels represent a cross section of application domains which greatly benefit from the large increase in throughput offered by manycore architectures.

7.3.1 Molecular Dynamics

We use the molecular dynamics package HOOMD (Highly Optimized Object Oriented Molecular Dynamics) [5] version 0.8. HOOMD is a general purpose molecular dynamics package that can take advantage of the computational power of GPUs using CUDA [21]. The two most computationally intensive functions in HOOMD are the Lennard-Jones potential computation and neighbor list generation, making up over 95% of the runtime. Note that HOOMD also supports other potentials, which all have the same computation and memory patterns as the Lennard-Jones computation.

The neighbor list function (NL) determines for every particle being simulated which other particles are close enough that their Lennard-Jones interactions with the current particle have to be taken into account. Since all particles move during the simulation time frame, the neighbor list is regenerated every 10 time steps. To avoid the need to check every particle against every other particle, particles are sorted into spatial bins in a preliminary step. Each particle then computes the distance between it and all particles in all neighboring bins, adding those particles that fall inside a cutoff radius to its neighbor list. To avoid regenerating the neighbor list each time step, the cutoff radius is made larger than necessary, so that particles which might move inside the real cutoff radius in several time steps are also added to the neighbor list.

The Lennard-Jones function (LJ) calculates the Lennard-Jones potential for each particle each time step, calculating distance and force for each particle on the neighbor list. Both kernels are parallelized by assigning each particle to a single thread.

We run a the standard HOOMD benchmark simulating a liquid consisting of 64000 particles at a packing fraction of 0.2 interacting via the Lennard-Jones force. We simulate the first 600 time steps.

7.3.2 DNA Sequence Alignment

We use the program MummerGPU [24] (SA), which uses a suffix tree to efficiently find alignments of short DNA se-

Structure	Fields per Entry	State per Entry	Number of Entries	Total Structure Size
Memory Divergence Table	PC, Thread Bitmask	32 bits + W bits	$M \cdot N$	$16 - 256$ bytes
Per-Thread Divergence Counter	Per-Thread Counter	8 bits	$W \cdot N$	$32 - 512$ bytes
Per-Warp Slip-Limit Bitmask	Thread Bitmask	W bits	N	$4 - 16$ bytes
Max-Slip Counter	Per-Core Counter	8 bits	1	1 byte

Table 1: New structures needed to support diverge on miss. N is the number of warps per core, which range from 1 to 16. W is the warp width, which is assumed to be 32 throughout this paper.M is the number of divergent memory operations per warp allowed, which is set to 2.

quences (such as those generated by high-speed DNA sequencing machines) against a reference genome. The tree is traversed from the root in a data dependent manner, with each edge holding a variable number of base pairs which must all match for the traversal to proceed to the next node.

Note that MummerGPU 1.1 has an inefficiency that we have fixed in our version. Originally each input string was stored contiguously in memory, making SIMD reads of the input inefficient, due to each SIMD lane's accessing memory locations which are separated by a stride. Following the example of HOOMD's neighbor list layout, we rearranged the input strings to be interleaved in memory, allowing the hardware to access them in a contiguous manner.

MummerGPU parallelizes its computation by mapping each input string to a thread. Similar to Schatz *et al.* [24], we run SA in the exact matching mode, matching batches of synthetic snippets of length 25, 50, 200 and 800 base pairs sampled randomly from the *Bacillus anthracis* genome ($GenBankID : NC\_003997.3$) to match against itself. Each batch contains a total of one million base pairs, with batches containing longer string containing linearly fewer samples. We report the average performance over all 4 string lengths.

7.3.3 Ray Tracing

We use the bwfirt ray tracing framework [23], and specifically the provided SimpleBVH ray tracer as our test application. SimpleBVH decomposes the scene into a bounding volume hierarchy tree. Each ray traverses the tree to find the object that it hits in the scene. Bwfirt uses SimpleBVH to do path tracing through a given scene, letting rays bounce around a scene multiple times until they hit a light source.

We parallelize SimpleBVH by having each thread trace a different ray through the scene. As our input we use the conference scene with approximately 1 million triangles and set the resolution of the generated image to 1024 by 1024 pixels.

7.3.4 Data Mining

We use the k-means program (KM) from Minebench [20]. The k-means code randomly generates N cluster centers, where N is given by the user. It then computes the distance between each point and each cluster center and assigns each point to the cluster with the closest center. After completing the reassignment of points to clusters it recomputes the cluster centers as the average of all points assigned to the cluster. The last two steps are repeated until the number of points switching cluster to another falls below a pre-specified threshold.

Both the distance computation per point and the recomputation of the cluster centers can be easily parallelized. We assign each point to a thread for the distance computation as well as the cluster center recomputation. We run k-means with 32 clusters and with the provided input set of roughly half a million data points, each with 36 features

7.3.5 Image Manipulation

We use a blurring kernel (GF), which computes the 3 by 3 Gaussian blur for each pixel of the input image. Each warp is assigned an image tile consisting of 32 by 32 pixels, with threads being assigned a single row in the tile. The input is a randomly generated black and white image with 2048 by 2048 pixels resolution.

8. EVALUATION

Our baseline for all comparisons unless otherwise stated is that each core has a single warp.

8.1 Application Behavior

We first explore the performance characteristics and scaling behavior of our selected kernels on the baseline chip as outlined in Section 7.2. Table 3 shows some of the most important performance aspects of each application, along with their flops and bandwidth usage with 1 and 16 warps per core with blocking SIMD execution.

Each of these kernels access their main data structure in their critical loop. Any cache miss by a load in the critical loop will cause a warp to almost immediately stall for all kernels, since the following instructions are data dependent on the load.[4] The number of instructions per memory op determines how sensitive to cache miss rate each kernel is. As an example, assume that all loads result in cache misses, so that each warp will have to stall after X instructions, where X is the number of instructions per memory op. If there are a large number of instructions per memory op, then only a small number of warps is needed to keep a core busy while a single warp is stalled waiting on memory. If the number of instructions per memory op is small, then a larger number of warps are needed. Thus, for a given number of warps per core, cores executing kernels with fewer instructions per memory op will stall more often and have lower performance. The same rule applies at any given cache miss rate.

Most of the kernels have high cache miss rates due to low temporal or spatial locality. The exception is k-means kernel, which has excellent temporal locality and a small working set per warp.

As a consequence, k-means is the only kernel which can exploit the full performance of the base chip with only a single warp per core, achieving 2 teraops/sec. The other kernels are all limited by memory stalls to much lower performance, with sequence alignment achieving only 2.9% of the maximum possible performance.

[4]Writes also stall warps quickly, but here the critical resource is the size of the write buffers.

Kernel Name	instructions per memory op	off-chip bandwidth (GB/sec)	inst per sec. (MInst/sec)
Neighbor List Generation (NL)	19	8.15/111.4	156/2048
Lennard-Jones Force Calculation (LJ)	25	26.34/256	204/1599
DNA Seq. Align. (SA)	7	62.77/256	57/445
Ray Tracing (RT)	15	30.60/256	124/767
K-Means (KM)	5	26/256	2048/634
Gaussian Filter (GF)	8	77.91/256	65/215

Table 3: Number of instructions per memory operation, bandwidth usage and instructions per second for each kernel. The BW and inst/sec data are presented as A/B, where A is the performance with a single warp per core and B is with 16 warps per core with no diverge on miss.

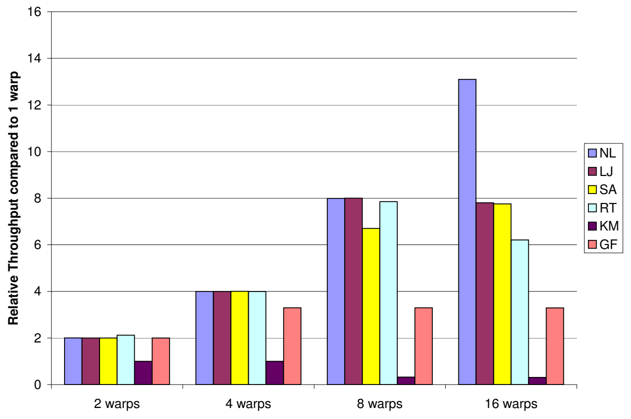

Figure 4: Increase in performance of 2 to 16 warps per core relative to a single warp per core. All configurations use blocking SIMD execution.

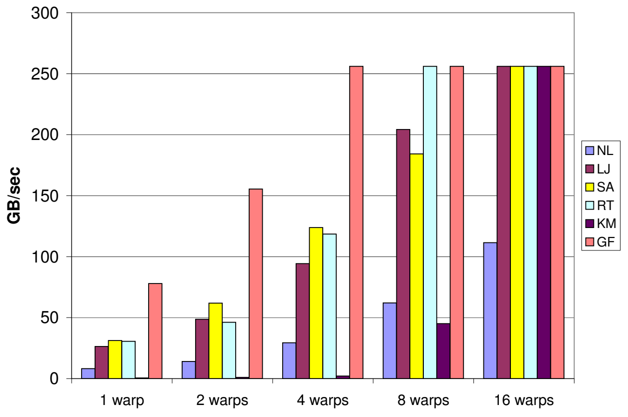

Figure 5: Bandwidth usage of all kernels with 1 to 16 warps per core. The total available bandwidth is 256 GB/sec.

Figure 4 shows the increase in throughput when we increase the number of warps per core from 1 to 16, and Figure 5 shows the bandwidth used for the same configurations. The neighbor list generation kernel shows the best increase, being limited by arithmetic throughput with 16 warps per core, as shown in Table 3. On the other hand, the sequence alignment and ray tracing kernels become bandwidth bound at 4 and 8 warps respectively. K-means suffers greatly from cache thrashing, as its performance decreases starting at 8 warps, due to the combined working set of all warps in each core starting to exceed the size of the cache. Ray tracing also suffers from cache thrashing at 16 warps.

8.2 Fixed Slip Performance

Figure 6 shows the relative speedup for 1 to 16 warps per core with a fixed maximum slip value compared to normal, blocking SIMD execution. We first determined which single fixed maximum slip value gave the best speedup across all kernels for each #warps/core configuration, shown as avg max slip in the Figure. We also determined the best fixed maximum slip value for each kernel and #warps/core and calculated the overall speedup using these individually best max slip values. This value is shown as best mix of max slip in the Figure. The difference at 1 and 2 warps is very significant, with relative speedups of of 2.65 vs. 4 at 1 warp per core and 2.88 vs. 3.15 at 2 warps per core. Moreover, the k-means kernel (which is ALU bound) exhibits a slowdown vs. blocking warps.

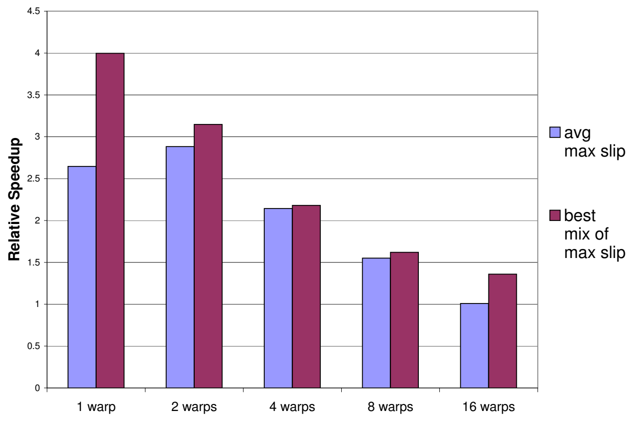

Figure 6: Speedup relative to blocking SIMD execution with 1 to 16 warps per core. We show diverge on miss and a fixed maximum slip across all kernels for one particular config versus combining the best fixed slip for each kernel.

These results show that the the maximum slip value cannot be set statically across all applications, but has to adapt to the workload.

8.3 Adaptive Slip Performance

Figure 7 shows the speedup with the diverge on miss and adaptive slip versus blocking warps for 1 to 16 warps per core broken out per kernel. Figure 8 shows the geometric mean speedup across all kernels for the same combinations of number of warps per core compared to a baseline of 1 warp per core and blocking SIMD execution.

Looking at the behavior of individual kernels shown in Figure 7 first, we see that the 2D Gaussian filter has the highest speedup of all tested kernels, increasing throughput to 8.3 times the speed of blocking SIMD execution at 1 warp per core. As the number of warps increases, the relative speedup compared to blocking SIMD execution decreases, as the kernel becomes bandwidth bound relatively quickly. The neighbor list generation and Lennard-Jones force calculation kernels also show high speedups at 2.5 and 5.6 respectively with 1 warp per core. As the number of warps per core is increased, these two kernels start to be limited by ALU throughput and bandwidth respectively, showing almost no speedup at 16 warps. The DNA sequence alignment kernel has its best speedup at 1 warp per core with 4.23. The alignment is also bandwidth bound as the number of warps increases.

K-means is a counterpoint to the other kernels, showing no appreciable speedup. This is because each thread reuses the data for the its point 32 times (once for each of the 32 cluster centers), leading to a small number of initial cache misses followed by the great majority of memory accesses hitting in the data cache. This behavior only changes at 8 and 16 warps, as the number of threads per core overwhelms the data cache and capacity misses result in a slowdown. With 8 warps, diverge on miss can provide a small speedup, as threads can reuse data in the cache in some cases where blocking warps would mean that the accesses would be too far apart in time. This is a good example how diverge on miss can help workloads which require a large number of warps for part of their execution, but are also limited by cache thrashing in other parts.

In overall performance the biggest gains can be seen for 1 to 4 warps per core, where the geometric mean speedup of diverge on miss with adaptive slip compared to blocking SIMD execution is 3.14 to 1.75 for 1 to 4 warps. Figure 8 shows that the speedup over 1 warp per core and blocking SIMD execution is 3.14, 4.66 and 5.38 for diverge on miss, and 1, 1.8 and 3.07 for blocking SIMD execution for 1,2 and four warps per core respectively.

At 8 and 16 warps per core many kernels become purely bandwidth or ALU bound, which means that benefit of adaptive slip decreases. Diverge on miss with adaptive slip improves overall performance by 26% at 8 warps per core and by 3.8% at 16 warps per core compared to blocking SIMD execution.

Compared to the scaling of performance for blocking SIMD execution shown in Figure 4, we can see that a core with 2 warps and diverge on miss and adaptive slip can provide equivalent performance to a core with 16 warps and normal SIMD execution. From the area estimates in Section 6.1, we can see that such a core is approximately 35% smaller than a core with 16 warps.

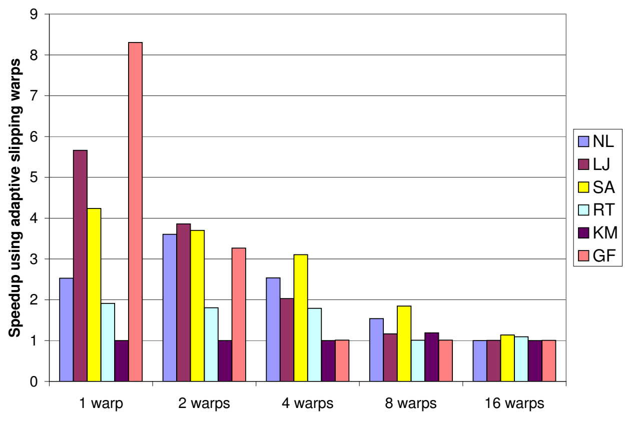

Figure 7: Speedup of adaptive slip versus blocking SIMD execution for different configurations. We sweep the number of warps per core from 1 to 16, showing the speedup due to adaptive slip at each.

Diverge on miss with adaptive slip control can also provide a higher peak performance (5.38 times the baseline) than normal execution (4.14 times the baseline), but only requires 4 warps per core versus 16 warps per core. Because diverge on miss can tolerate more latency with a given number of warps, it is more limited by bandwidth constraints. As such, area saved by smaller cores could be used for more I/O, bigger caches or other structures which reduce off-chip bandwidth.

8.4 Cores with Private L2 Caches

Since both Larrabee and Niagara provide unified second level cache on-chip, we also explore whether adding diverge on miss to a design where the SIMD cores are coupled to L2 caches is worthwhile. We simulate a design with the same number and type of cores as in previous experiments, but where each core has a private 256KB L2 cache. Each L2 cache is has 32B cache lines and is 16-way set associative. We also keep off-chip bandwidth and access latency constant.

The mean performance of such a chip compared to a chip without L2 caches (data not shown in the Figures) is 8.1%, 7.2%, 7.4%, 52.4% and 115.9% higher for 1 to 16 warps per core. Performance scaling also improves with the addition of L2 caches. The relative performance of a core with 16 warps compared to 1 warp per core is 6.35 with L2 cache and 4.14 without.

Figure 9 shows the speedup for each kernel with the diverge on miss and adaptive slip versus blocking warps for 1 to 16 warps per core. Figure 10 shows the geometric mean speedup of diverge on miss and blocking SIMD execution as the number of warps per core is increased from 1 to 16 compared to 1 warp per core and blocking SIMD execution.

The overall performance increases are even larger than in Figure 8 primarily because kernels are less bandwidth bound and have fewer L2 caches misses, so that the misses which can be hidden with diverge on miss cover a larger fraction of misses and provide a larger relative improvement in performance. Adaptive slip can provide a higher peak performance of 7.36 times the base performance versus 6.35 for blocking SIMD execution, which needs twice as many

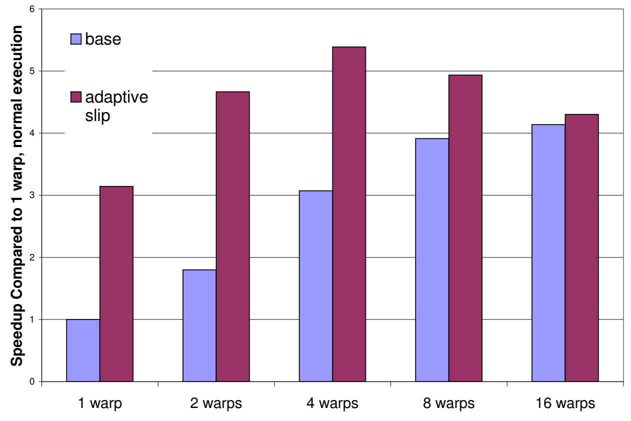

Figure 8: Comparing the speedup of both blocking SIMD execution and adaptive slip from 1 to 16 warps. The baseline is 1 warp per core with normal execution. Adaptive slip can provide a higher peak performance of 5.38 times the base performance versus 4.14 for normal execution, which needs 4 times more warps.

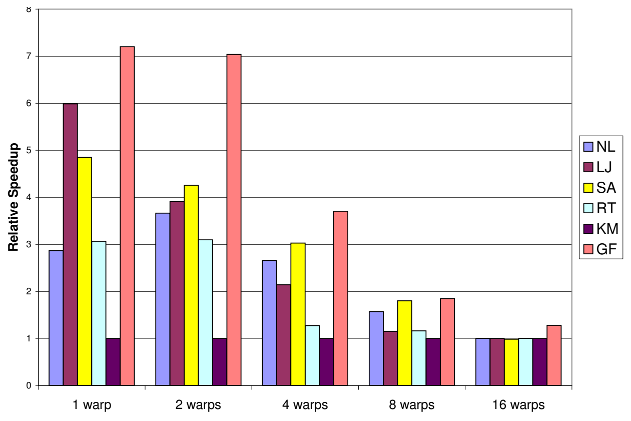

Figure 9: Speedup with adaptive slipping warps versus blocking SIMD execution with each core having a private 256KB L2 cache. We sweep the number of warps per core from 1 to 16, showing the speedup due to adaptive slip at each.

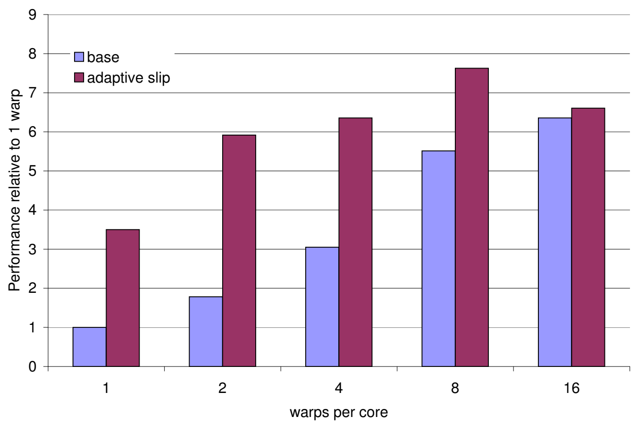

Figure 10: For cores with 256 KB L2 caches, we compare the speedup of blocking SIMD execution and adaptive slipping warps from 1 to 16 warps. The baseline is 1 warp per core with blocking SIMD execution.

warps. Adaptive slip needs only 4 warps per core to provide equivalent performance to normal execution at 16 warps.

9. CONCLUSIONS

To maximize performance within power and area constraints, designers have turned to architectures with many small, multithreaded SIMD cores for throughput oriented workloads. Such architectures work well for applications with regular data access patterns, but can easily become latency bound for workloads with more complicated scatter/gather access patterns.

We introduce the concept of diverge on miss, which allows SIMD warps to continue execution even when a subset of their threads are waiting on memory. This provides benefits when runahead threads prefetch cache lines for lagging threads. It also increases throughput when divergent threads experience misses and runahead and lagging threads continually leapfrog each other, rather than continually being held back by the slowest thread. The key insight is that SIMD cores' support for branch divergence can be elegantly extended to support memory divergence .

We show that on a set of data-parallel kernels, diverge on miss can provide speedups as high as 3.14 over normal SIMD execution or reduce the core area by 35% at constant performance. It can also provide 30% higher absolute peak performance than normal execution with fewer warps per core.

10. ACKNOWLEDGMENTS

This work has been supported in part by NSF grants IIS-0612049, CNS-0615277, and CNS-0509245, and a gift from NVIDIA Research. We would also like to thank Timo Aila and the anonymous reviewers for their detailed, constructive suggestions on how to improve the paper.

11. REFERENCES

[1] Intel Advanced Vector Extensions Programming Reference, 2009. http://software.intel.com/file/21558.

[2] K. Aingaran, P. Kongetira, and K. Olukotun. Niagara: A 32-way Multithreaded Sparc Processor. *IEEE Micro*, 25:21–29, 2005.

[3] AMD. ATI CTM Guide: Technical reference manual. Technical report, AMD, 2006. Version 1.01.

[4] AMD. ATI Radeon HD 2900 Technology, GPU Specifications, 2007.

[5] J. A. Anderson, C. D. Lorenz, and A. Travesset. General Purpose Molecular Dynamics Simulations fully implemented on Graphics Processing Units. *J. of Computational Physics*, 227(10):5342–5359, 2008.

[6] R. D. Barnes, E. M. Nystrom, J. W. Sias, S. J. Patel, N. Navarro, and W.-m. W. Hwu. Beating In-Order Stalls with "Flea-Flicker" Two-Pass Pipelining. In *Proc. 36th IEEE/ACM Int'l Symp. Microarchitecture (MICRO '03)*, pages 387–398, 2003.

[7] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture based on Single-Chip Multiprocessing. In *Proc. 27th Int'l Symp. Computer Architecture (ISCA '00)*, pages 282–293, 2000.

[8] D. Blythe. The Direct3D 10 system. *ACM Trans. Graphics*, 25(3):724–734, 2006.

[9] M. Boyer, D. Tarjan, S. T. Acton, and K. Skadron. Accelerating Leukocyte Tracking using CUDA: A Case Study in Leveraging Manycore Coprocessors. In *Proc. 24th Int'l Parallel and Distributed Processing Symp. (IPDPS '09)*, pages 1–12, 2009.

[10] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Trans. on Graphics*, 23(3):777–786, 2004.

[11] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A Performance Study of General-Purpose Applications on Graphics Processors using CUDA. *J. of Parallel and Distributed Computing*, 68(10):1370–1380, 2008.

[12] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J.-H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi. Merrimac: Supercomputing with Streams. In *Proc. 15th ACM/IEEE Conf. Supercomputing (SC '03)*, page 35, 2003.

[13] B. K. Flachs, S. Asano, S. H. Dhong, H. P. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. S. Liberty, B. W. Michael, H.-J. Oh, S. M. Müller, O. Takahashi, K. Hirairi, A. Kawasumi, H. Murakami, H. Noro, S. Onishi, J. Pille, J. Silberman, S. Yong, A. Hatakeyama, Y. Watanabe, N. Yano, D. A. Brokenshire, M. Peyravian, V. To, and E. Iwata. Microarchitecture and Implementation of the Synergistic Processor in 65-nm and 90-nm SOI. *IBM J. Research and Development*, 51(5):529–544, 2007.

[14] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proc. 40th IEEE/ACM Int'l Symp. Microarchitecture (MICRO '07)*, pages 407–420, 2007.

[15] M. Garland, S. L. Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel Computing Experiences with CUDA. *IEEE Micro*, 28(4):13–27, 2008.

[16] A. Glew. MLP yes! ILP no! In *ASPLOS Wild and Crazy Ideas*, 1998.

[17] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, 2008.

[18] G. H. Loh. The Cost of Uncore in Throughput-Oriented Many-Core Processors. In *Workshop on Architectures and Languages for Throughput Applications*, 2008.

[19] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors. In *"Proc. 9th Int'l Conf. High Performance Computer Architecture (HPCA '03)"*, pages 129–140, 2003.

[20] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. MineBench: A Benchmark Suite for Data Mining Workloads. In *Proc. 2006 IEEE Int'l Symposium on Workload Characterization (ISWC '06)*, pages 182–188, 2006.

[21] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[22] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for a Single-Chip Multiprocessor. In *Proc. 7th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 2–11, 1996.

[23] M. Raab, L. Grünschloss, J. Hanikaz, M. Finckh, and A. Keller. bwfirt. http://bwfirt.sourceforge.net/.

[24] M. Schatz, C. Trapnell, A. Delcher, and A. Varshney. High-Throughput Sequence Alignment using Graphics Processing Units. *BMC Bioinformatics*, 8(1):474, 2007.

[25] L. Seiler et al. Larrabee: A Many-Core x86 Architecture for Visual Computing. *ACM Trans. on Graphics*, 27(3):1–15, 2008.

[26] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual Flow Pipelines. In *Proc. 11th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, pages 107–119, 2004.