

# Improving SIMT Efficiency of Global Rendering Algorithms with Architectural Support for Dynamic Micro-Kernels

Michael Steffen and Joseph Zambreno  
*Electrical and Computer Engineering*  
*Iowa State University*  
*Ames, IA, USA*  
{*steffma, zambreno*}@iastate.edu

**Abstract**—Wide Single Instruction, Multiple Thread (SIMT) architectures often require a static allocation of thread groups that are executed in lockstep throughout the entire application kernel. Individual thread branching is supported by executing all control flow paths for threads in a thread group and only committing the results of threads on the current control path. While convergence algorithms are used to maximize processor efficiency during branching operations, applications requiring complex control flow often result in low processor efficiency due to the length and quantity of control paths. Global rendering algorithms are an example of a class of application that can be accelerated using a large number of independent parallel threads that each require complex control flow, resulting in comparatively low efficiency on SIMT processors. To improve processor utilization for global rendering algorithms, we introduce a SIMT architecture that allows for threads to be created dynamically at runtime. Large application kernels are broken down into smaller code blocks we call  $\mu$ -kernels that dynamically created threads can execute. These runtime  $\mu$ -kernels allow for the removal of branching statements that would cause divergence within a thread group, and result in new threads being created and grouped with threads beginning execution of the same  $\mu$ -kernel. In our evaluation of SIMT processor efficiency for a global rendering algorithms, dynamic  $\mu$ -kernels improved processor performance by an average of  $1.4\times$ .

## I. INTRODUCTION

Single Instruction, Multiple Thread (SIMT) processors offer an attractive alternative platform for parallel computing by supporting a large number of on-chip processor cores and even larger numbers of parallel threads. To support these high core counts, SIMT architectures impose additional structure on multi-threaded applications and introduce some simplifications to conventional processor datapaths that can negatively impact performance. These limitations on program structure often require new algorithms to be developed for improving SIMT performance; making the porting process for existing algorithms to SIMT difficult. Similarly, while many algorithms have been developed for SIMT processors, performance is often less than the expected Amdahl’s speedup [1], mainly for architectural reasons.

Consider, for example, physically-based global rendering algorithms, that seek to produce highly realistic images

through the modeling of the physics of light transport [2]. Conceptually, these global rendering algorithms map nicely to wide SIMT hardware, since individual pixels can be represented by a single thread, resulting in thousands of individual threads with no inter-thread data dependencies. These threads can then be mapped to one of the many parallel cores that can switch between threads at minimal cost [3], [4]. In practice, however, global rendering algorithms require large amounts of memory bandwidth as well as complex scalar thread flow (i.e. branching). Results from SIMT execution on NVIDIA GPUs show that performance is typically limited by the complex control flow and not the memory bandwidth requirements [5].

Performance degradation from SIMT branching is a consequence of multiple cores being required to execute the same instruction in lock-step. As will be further explained in Section II, threads running on processors that execute the same instruction are part of a group (referred to as a warp) that is defined at application launch. In our observations of branching performance, scalar thread branching alone is not the cause of performance loss, but scalar threads executing in SIMT as a warp can affect performance when control flow diverges amongst the threads. Our initial experiments indicate that for global rendering algorithms, the loss in efficiency due to branch divergence is as high as 65% (see Section VII).

These rendering algorithms are complex by nature; whether or not the kernels are written for a conventional graphics pipeline or entirely as compute applications, branching is a critical operation that allows for additional realistic image effects. As rendering algorithms continue to evolve by implementing more realistic physically-based methods, the efficiency of data-dependent looping operations will determine the performance for rendering an entire image. As a result, additional overhead in optimizing complex rendering algorithms for SIMT will be required to achieve high performance [5], [6].

In this paper, we propose to allow threads that do not require any explicit synchronization to dynamically spawn new threads at runtime. The locations in a kernel where a branching statement can lead to a significant code diver-

gence is where what we call a  $\mu$ -kernel can be potentially spawned with dynamically created threads. Dynamically created threads can then execute any  $\mu$ -kernel, which are essentially subsections of the original application kernel. The advantage for creating new threads is simply that the performance loss from branching can be avoided.

Our main contributions include:

- A new branching SIMT architecture that supports dynamic thread creation during runtime for parallel threads.
- A hardware component design that schedules dynamically created threads for improved SIMT processing efficiency.
- A software implementation utilizing dynamic  $\mu$ -kernels for increasing processor utilization of a global rendering algorithm.

Our initial simulation results using dynamic  $\mu$ -kernel execution indicates that processor efficiency increases by  $1.9\times$  for a global rendering algorithm. This increase in efficiency results in the application being able to compute an average of 67 million rays per second compared to 47 million rays per second using traditional SIMT hardware. The theoretical performance of dynamic  $\mu$ -kernels also lowers the gap between SIMT and the Multiple Instruction, Multiple Data (MIMD) theoretical ideal to  $0.6\times$ .

The remainder of this paper is organized as follows: Section II provides an overview of the SIMT architecture used in this paper. Section III discusses global rendering algorithms. Section IV describes in detail our hardware architecture for creating and scheduling new threads. A software implementation using dynamic threads for global rendering is in Section V. Section VI details our experimental setup and Section VII presents experimental results. Section VIII discusses previous related research and Section IX concludes the paper with a discussion of planned future work.

## II. SIMT ARCHITECTURE

SIMT processors are designed for executing large numbers of parallel threads that are defined at the start of execution and run the same application kernel [7]. The overall runtime for all threads (from the first thread launch to the last thread completion) is the critical performance metric, rather than individual thread runtime. With this metric in mind, these architectures are tailored to high-throughput parallel processing with scheduling policies that focus on keeping processor ALUs active. Parallel processing is accomplished by having a large number of lightweight cores that lack branch prediction, out-of-order scheduling, and other scalar thread acceleration hardware. Individual cores are kept from stalling by constantly switching between all available threads. While switching between threads increases the individual runtime for a thread, it allows for high latency instructions that would stall a processor to be tolerated by executing instructions from other active threads.

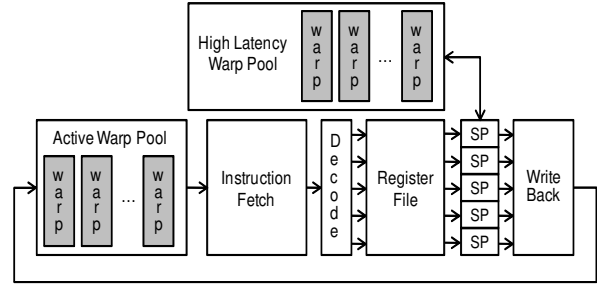


Figure 1. Single streaming multiprocessor architecture composed of multiple stream processors.

For area and energy efficiency, SIMT processors are grouped together to share certain hardware components, such as register files and instruction fetch units [8]. SIMT architectures commonly define processors using a hierarchy. At the top level, the architecture is composed of an array of processors referred to as Streaming Multiprocessor (SMs) [9]. All SMs have access to multiple memory controllers through a networked interconnect to allow for highly banked parallel memory operations. SMs operate in isolation and communication is not supported amongst SMs. As a result, two threads assigned to different SMs have no synchronization support, whereas threads assigned to the same SM have some limited synchronization capabilities.

The second level of the SIMT processor hierarchy is inside each SM component. SMs are composed of multiple Stream Processors (SPs) [9] that execute scalar threads. SPs are primarily ALUs and do not have individual register files or instruction fetch units; instead, SPs receive instructions and data from a single shared instruction fetch unit and register file. The SM register file is highly banked to allow for multiple simultaneous accesses by SPs. Each SM also contains two thread queues to manage the threads assigned to it (see Figure 1).

For all SPs inside of an SM to share a instruction fetch unit effectively, threads running within a processor group must execute the same instruction, and register names are manipulated to access different data on all the SPs. Individual threads are then grouped into warps [9] at application launch and remain together throughout their lifetime. The number of threads in a warp can be a multiple of the number of SPs in a SM. All threads in a warp then execute in lock-step, requiring only one instruction fetch for all threads. Warps are the granularity used for scheduling inside an SM, where the individual threads of a warp are assigned an individual SP.

Unlike conventional scalar thread processing, SPs execute one instruction from a warp and can then switch to another warp on the next cycle. Fast switching is done by using a scheduling thread queue (with each element organized as warps) and a large register file. The register file is large enough to accommodate all threads assigned to an SM. The number of registers that can be used per thread is flexible,

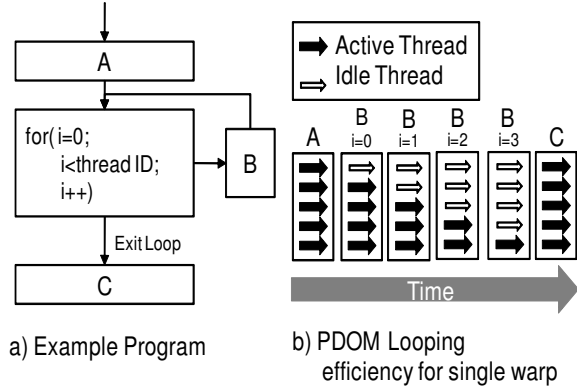


Figure 2. PDOM branching efficiency for a single warp performing a looping operation.

but can be the limiting factor in terms of the number of threads assigned per SM (the size of the thread queue is also another possible limiting factor). A warp is fetched from the thread queue after execution of the current warp instruction. Once a warp has finished data write-back for the fetched instruction, the warp is placed back in the thread queue. If the current executed instruction requires a high-latency operation before write-back, the warp is placed into another thread queue. Once the instruction finishes, the warp is moved back into the scheduling thread queue.

To allow for branching instructions, threads within a warp must be allowed to follow different control flow paths while using a single instruction fetch unit. Consequently, all possible control paths in a warp are executed sequentially and threads not requiring the current control path do not commit the results of those instructions. To minimize the performance loss of SPs sitting idle, reconvergence algorithms, such as post-dominator (PDOM) [10], are implemented to schedule all control paths with minimal processor idle time. Figure 2 shows an example for a simple data dependent looping operation (similar to the one presented by Fung et al. in [11] for instruction branching), and how PDOM results in multiple idle streaming processors. The example application in Figure 2a only has two control paths for the loop (running B again or executing C) and a single thread convergence location (C). PDOM first executes the first control path for B until there are no more threads requiring this path (Figure 2b). The next control path is executing C, where all threads would be enabled, since C is the convergence location for all the threads in the warp. If the runtime for B is much larger than for both A and C, the looping operation would only be 50% efficient since only half of the SPs would be used on average.

Consequently, applications that require complex control flow or long-running diverging branches can have decreased processor efficiency, due to the idle SPs that are completing all control flow paths before the warp can converge from

### Example 1 Traditional Ray Tracing Kernel

```

1: while ray is not finished do
2:   while node is not a leaf do
3:     if pass through multiple child nodes then
4:       push nodes on stack
5:     end if
6:     traverse tree to next node
7:   end while
8:   while untested objects do
9:     ray-object intersection test
10:  end while
11:  pop stack
12: end while

```

the diverging paths. Improving the under-utilization of SPs can have a dramatic effect on the performance of an application, without the need for additional processing power. As previously mentioned, in our observation of branching performance, scalar thread branching only decreases performance when threads in a warp follow different control paths. If all threads in a warp follow the same control flow, there is no performance loss. Additionally, two different warps can have diverging control flow with no performance loss, presuming all threads in the warp follow the same path.

We propose that, instead of allowing threads in a warp to diverge at a branching instruction, that new threads be created to execute the required instructions for the control path taken by those threads. Once the new threads have been created, the current running threads can then exit to prevent the warp from executing diverging control paths. New threads that begin execution at the same  $\mu$ -kernel are then placed into the same warp. This process results in multiple warps that are executing different instructions (but have no performance loss); threads inside a warp are collected to follow the same control path.

### III. GLOBAL RENDERING ALGORITHMS

Global rendering is a class of algorithms that use entire scene data to create an image. Typical implementations model parts of the physical world to create a realistic image. While physically-based computations require higher computational power than local rendering algorithms, the results are often more realistic and special effects are built into the algorithm, resulting in improved user interaction experiences and easier code development.

An important characteristic in the design of a rendering algorithm is the runtime to generate a single image. Both off-line and real-time algorithms will target a fixed amount of time that can be tolerated for creating a single image. For real-time interactive graphics, the tolerated time is based on the human visual system, oftentimes targeting 1/30 of a second. Off-line rendering has more tolerance in the rendering time, and these implementations often target from a few seconds to a few days to create a single image. Once a time limit has been established, developers design

an algorithm for the highest image fidelity as possible. In general, the more computational power available to the algorithm, the higher the quality of the image, given the same time constraints. Computational power is determined by the computer hardware and the algorithm’s efficiency on the hardware. By improving the efficiency of global rendering algorithms on SIMT processors such as GPUs, the system can improve image realism under the same time constraints.

### A. Ray Tracing

In this paper we focus on a specific technique used in many global rendering algorithms called ray tracing [12]. Ray tracing is used to gather global data in the scene, and rendering algorithms that use ray tracing typically result in large rays per pixel ratios. Some example usages include:

- Determining if an object being drawn to the screen is in a shadow of another object. A ray is created for each pixel used to draw the object, with the origin set to the object surface being drawn by the pixel and the direction pointing to a light source. If the ray intersects another object before the light source, the pixel drawing that object should be drawn with a shadow.
- Rendering reflections off an object’s surface. A ray is created for each pixel used to draw a reflective surface to the screen. The origin of the ray is set to the object surface and the direction of the ray is calculated using the angle of reflection from the camera. The first object that the ray intersects is then used to render the reflected surface.
- Global illumination from light emitting off an object surface. To determine which objects are lit by this surface, multiple rays are created with origins set to the object surface, with ray directions that can be randomly generated. Objects that rays intersect are then lightened by the illuminated surface.

Determining the first object that intersects a ray becomes a searching problem among all scene objects. To accelerate this search, tree data structures are used, such as a *k*-tree [13] or Bounding Volume Hierarchies [14]. Leaf nodes of the tree represent a small portion of the total spatial volume of the entire 3D scene and contain all the objects within that space. The tree nodes are used to subdivide a larger spatial representation into multiple smaller spatial areas. Node subdivision is performed until leaf nodes contain a specified number of objects. Rays are the input into the tree and are traversed down from the root node to leaf nodes that the ray passes through, visiting nodes in the order that the ray moves through the 3D world. An example tree traversal algorithm is shown in Example 1.

### B. Ray Tracing on SIMT Architectures

Conceptually, the large number of rays required to render a scene is well-suited to implementation on SIMT proces-

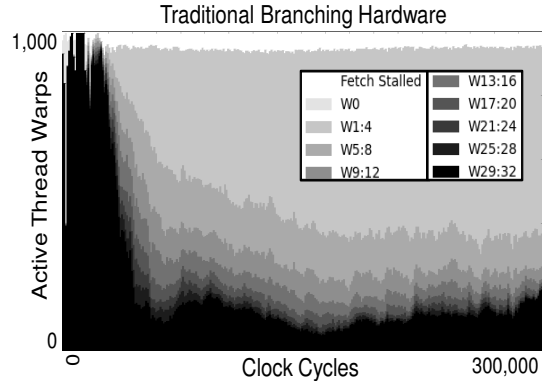


Figure 3. Divergence breakdown for warps using traditional SIMT branching methods for the `conference` benchmark. Higher values in the key represent more threads that are active within a warp. This figure was created using AerialVision [15].

sors, since each ray can be processed in parallel and each thread follows the same algorithm for traversing the tree data structure. The complex control flow for global rendering algorithms comes from the tree traversal algorithm. To traverse the tree with a ray requires three loops. The first loop (line 1) of Example 1 is used to guarantee that all relevant tree leaf nodes are tested before the ray is finished being processed, since a ray can pass through multiple leaf nodes. The middle two loop statements are used to traverse down the tree from node to node until it finds a leaf node (line 2), and to test all objects inside the leaf node (line 5). Diverging control flow accrues for this algorithm due to the different number of loop iterations required by each ray for all three loop instructions. While rays traversing different paths down the tree do not cause any diverging control flow path since they are running the same instructions, the tree depth for each path is different. The number of objects stored in each leaf node may be different and the number of leaf nodes that a ray passes through also varies.

Since conventional SIMT hardware requires all threads in a warp to run the same instruction, threads within a warp that finish a loop run idle until all threads in the warp are finished as well. The overall result is that the number of clock cycles for any ray in a warp to complete is equal to the longest ray in the warp. Figure 3, plotted using AerialVision [15], illustrates how many streaming processors are running idle per clock for executing these nested loop iterations. This plot categorizes a warp into 10 different categories based on the number of threads in a warps that are not idle. Category W29:32 is the number of warps that have 29 to 32 active threads in the warp. Category W1:4 indicates that there are only 1 to 4 active warps and that the remanding are idle due to branching.

To improve the efficiency and performance for this algorithm (and others similar), we propose using dynamic  $\mu$ -kernels to remove the looping iterations that cause low

efficiency.  $\mu$ -kernels are created for instructions in a loop and new threads are created to perform the required instructions. Threads that would require looping will create a thread that calls the same  $\mu$ -kernel. After a  $\mu$ -kernel has finished executing and has created additional threads, the current thread will exit, since its child thread will continue the required computations. To utilize SIMT technology, newly created threads will then be placed into new warps with other threads that will execute the same  $\mu$ -kernel. The diverging control flow that resulted from the looping operations is removed since threads are no longer executing loop iterations, and are instead creating new processing threads that are organized for high processor efficiency.

#### IV. DYNAMIC $\mu$ -KERNEL ARCHITECTURE

Our dynamic  $\mu$ -kernel implementation is broken down into two components. The first part allows SPs to create new processing threads. The second part takes new threads and creates new warps that will not result in large divergent control paths. The process of creating threads at runtime and forming new warps is outlined in Figure 4. Threads are able to initiate the creation of threads inside an SP using a new instruction which we call *spawn*. The SM then groups the new threads with previously created threads that share the same targeted  $\mu$ -kernel in the partial warp pool. Once enough threads are grouped to form a new warp, the warp is placed into a new warp FIFO and waits to be scheduled for execution. When a currently scheduled warp finishes, a new warp from the warp FIFO is issued using the same resources as the finished warp and placed into the active warp pool.

In addition to the SM hardware for issuing new threads, data must be passed from the parent thread to the newly spawned child thread that is intended to continue the work of the parent thread. Data required for the spawned threads cannot be passed using registers, since the new thread is likely to be assigned a different SP than its parent. Since rendering algorithms typically require little persistent memory to define the current state of a thread, register states are instead saved to on-chip shared memory. When a thread is created at runtime, a memory pointer is provided to the thread allowing the thread to access its associated data.

##### A. Memory Organization

Similar to NVIDIA CUDA [9] supported architectures, the available memory spaces for a thread consists of registers, shared memory, local thread memory, global device memory, constant memory, texture memory, and a new memory space called *spawn memory*. Registers and shared memory are located on-chip and are tied to an SM. Local thread memory is stored in off-chip device memory, and is reserved for register overflow to reduce register counts for kernels and also for any intermediate data storage that is too large for on-chip memory. Constant memory and texture memory also use device memory and are used similar to local memory,

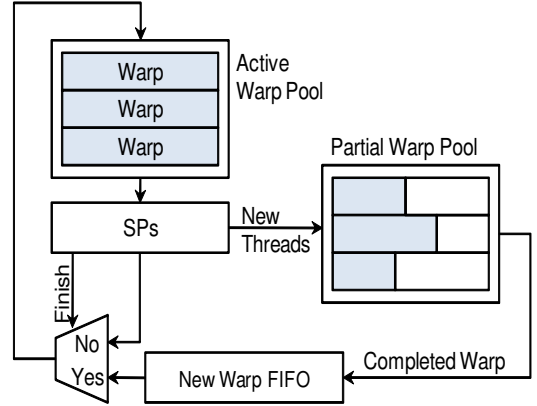


Figure 4. Dynamic thread creation hardware overview. New threads created by the SPs are placed into new warps waiting in the partial warp pool. Once enough threads have been created to complete a warp, the warp can replace an existing warp that has finished.

except that they are read-only and can be cached. Global device memory is off-chip and is shared across all SMs, and can also be allocated and accessed by a host processor.

Spawn memory may be implemented in on-chip memory inside an SM or device memory. The memory space is used for two purposes: storing the data to be passed between threads and storing partial warps during warp formation (see Section IV-C). This memory space is allocated at kernel launch time, since the size requirements can be computed off-line and are constant throughout application execution.

1) *Thread Usage of Spawn Memory*: The first section of this memory space is for storing data to be passed between threads. The allocation size is computed by the size of the data structure used for passed data between threads, multiplied by the number of threads that can be assigned to an SM. Since the size of the data structure may fluctuate depending on what  $\mu$ -kernel is being called, the largest data structure is used for the computation. This also requires that the  $\mu$ -kernels be predefined before the application is executed. Individual threads in an SM are then assigned a region of this memory space.

The method that threads use to access their spawn memory space depends on how a thread is created and when it is scheduled. However, all threads use a special register called *spawnMemAddr* to determine the spawn memory address. Threads initially created and scheduled at application launch have their *spawnMemAddr* set to a unique address inside the spawn memory using the equation  $SpawnMemoryBaseAddress + threadID * sizeof(DataToBePassedBetweenThreads)$ . Dynamically created threads are provided a memory pointer in the *spawnMemAddr* when scheduled that is used to obtain the appropriate spawn memory address (see Section IV-D). Threads that were created at application launch but were not able to be scheduled due to resource availability require

a spawn memory address that has been freed by a thread, made available when a thread exits from the last  $\mu$ -kernel.

Data in the spawn memory space is what is passed between a parent and child thread. If a thread is creating a child, the parent thread will store its current state in the spawn memory before calling the spawn instruction. If the thread is a child, the spawn memory space is used to retrieve data from its parent thread. Child threads can reuse the same spawn memory address to pass data to further children.

2) *Partial Warp Formation*: The second half of the spawn memory space is for storing dynamic threads during warp formation. The hardware components for creating new warps require consecutive memory addresses to store the metadata of individual threads belonging to a new warp. The number of consecutive memory address is equal to the number of threads in a warp. The minimal size required for this memory is a function of the number of threads that can be assigned to an SM, the number of threads per warp, and the number of  $\mu$ -kernels ( $size = NumThreads + (SpawnLocations - 1) * WarpSize$ ). The size allocated in memory is doubled to prevent a new thread's metadata from clobbering active threads. The spawn memory used for passing data between threads does not need to be doubled, since registers can be used to save original data when reusing the memory space for creating a child thread.

### B. Spawn Instruction

Spawn instructions take two parameters: an assembly code label (converted by the assembler to the Program Counter (PC) value), used to indicate the new thread's  $\mu$ -kernel, and a register that contains the memory pointer to the thread's spawn memory space. The spawn instruction performs two key functions. First, it updates the warp creation hardware used to assign the new threads into warps. Second, it performs a memory write operation that is required by the warp creation hardware to save the thread's metadata.

### C. Warp Formation

Figure 5 shows the architecture for the spawn instruction for an SM. The first operation performed is thread classification, where new threads are placed into warps. The PC value of the spawn instruction will be the same value for all threads executing the spawn instruction. The value is the same for all instructions since the PC value is statically compiled into the instruction and all instructions in a warp execute in lock-step. The PC is used in a look-up table (LUT) to determine the address in the spawn memory space where similar threads are being grouped into new warps. The functionality of the LUT is identical to the dynamic warp formation concept presented in [11], and provides the mechanism for taking a new thread's PC and providing an index to start forming new warps. The LUT is an on-chip memory organized as a fully associated cache where the number of entries is equal to the number of supported  $\mu$ -kernels. The content of our

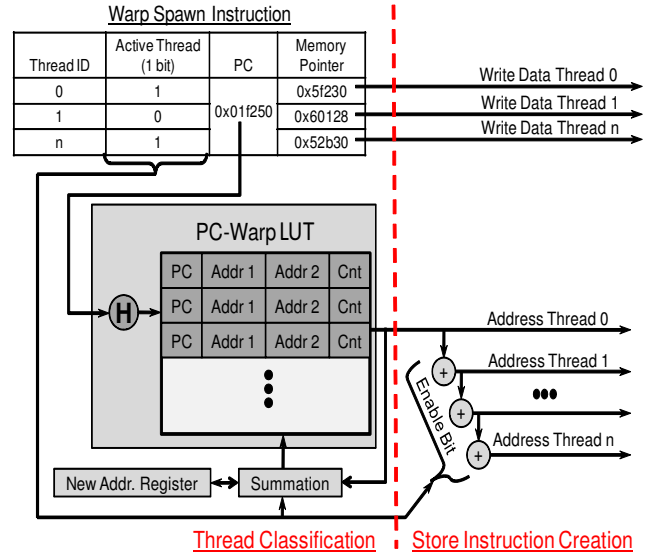


Figure 5. Architecture for the spawn instruction. Dynamically created threads identify existing threads that will follow the same control path using a look-up table. Once the warp has been identified for the new threads, the pointers are stored in memory for later use.

LUT is different from that of [11]; two memory addresses and one counter variable are stored for each line in the LUT. The counter keeps track of the number of threads already contained in the partially created warp. The first memory address is the spawn memory address where the current warp is being created. The second memory address is the overflow address used for creating the next warp for the same PC. A single spawn instruction may result in more threads being created for a  $\mu$ -kernel than the current warp being formed has room for. In this case the second memory address is required to create a second warp.

After the memory read from the LUT, the resulting first memory address and counter are both incremented based on the number of new threads (summation hardware). The number of threads is the sum of all active threads in the warp that executed the spawn instruction. Once incremented, both results are stored back into the LUT. If the counter is incremented over the size of a warp, the second memory address is incremented based on the overflow from the first memory address and replaces the first memory address. The second memory address is then set to the next available free memory address stored in a register. If the first memory address does overflow, this signals that a new warp has been created and is pushed into the new warp FIFO. The value pushed into the FIFO is the first memory address from the LUT that points to the spawn memory space containing the last thread in the finished warp.

The second operation is creating a store instruction to save the new thread's metadata into memory. The two addresses read from the LUT are used to compute a unique address

for all threads executing the spawn instruction. Since not all threads of a warp are active, memory addresses used for the store instruction are not computed for all threads. The memory addresses from the LUT are pipelined to each thread channel and incremented if that thread channel is enabled. The result is that all active thread channels have a memory address that is both unique and sequential. Threads in the warp that are not executing the spawn instruction result in a duplicated memory address, since the addition operation was disabled and the value is pipelined through the thread’s channel to feed additional thread channels. Since the threads are not active, they will not perform the store instruction and ignore the duplicated memory address. In the event that a warp is filled before the end of all active threads, the second memory address is used to start forming a new warp. After a memory address has been computed for each thread, a memory store transaction is generated for storing the memory pointer to the new thread’s data being passed by the parent thread to the second half of the spawn memory.

#### D. Scheduling

Once a warp has been defined in memory, the scheduler attempts to schedule the warp for execution. Similar to scheduling threads defined at kernel launch, SM resources (such as registers and the number of threads that can be stored in the SM thread queue) need to be available before beginning execution of a warp. To reduce the size of the dynamic warp FIFO, dynamic threads are given priority for scheduling over unscheduled threads defined at kernel launch. If not all SM resources are used during kernel launch, dynamically created threads can use the remaining available resources to be scheduled for execution. When currently executing warps exit, their resources are freed, which makes scheduling possible for additional warps.

Since spawn instructions can invoke multiple  $\mu$ -kernels, dynamically created threads may require different amounts of SM resources. To allow for easier scheduling, all threads use the maximum resource per category required by each of the  $\mu$ -kernels. This allows for any warp to replace an existing warp; the scheduler does not need to keep track of different warp resources. The tradeoff for this method is a decrease in the number of threads that can be actively executing if  $\mu$ -kernels are not balanced.

To give each thread in a warp access to its data stored in spawn memory, the *spawnMemAddr* special register is set to the memory address pointing to the data in spawn memory. The spawn memory pointer is stored in memory by the thread grouping hardware during the spawn instruction. The memory address provided to new threads in *spawnMemAddr* is computed from the memory address from the LUT. The individual thread values are computed by taking the address from the LUT that was used to store the last thread in a warp metadata to memory, and subtracting the thread ID inside

Processor Cores	30
Warp Size	32
Stream Processors per Warp	8
Threads / Processor Core	1024
Thread Blocks / Processor Core	8
Registers / Processor Core	16384
On-chip Memory / Processor Core	64 KB
Spawn LUT Size / Processor Core	1024 Bytes
Memory Modules	8
Bandwidth per Memory Module	8 Bytes/Cycle
L1 and L2 Memory Caching	None

Table I  
CONFIGURATION USED FOR SIMULATION.

the warp. Using this method, the size of the on-chip FIFO is reduced by a factor equal to the number of threads in a warp, since the individual thread address can be computed for a single value for the entire warp.

Scheduling new warps only when the warp is completely filled can cause threads to stall during thread formation due to a lack of remaining threads to finish the new warp. To prevent this problem, partial warps can be forced out of the thread pool and scheduled as incomplete warps. Threads are forced out only when the scheduler runs out of available warps to schedule. This happens only near the end of the application when all warps defined at launch time have finished and no new warps remain in the dynamic warp FIFO. The warp that is forced out is selected by the PC address of the  $\mu$ -kernel, starting with the lowest PC address.

#### V. PROGRAMMING MODEL FOR DYNAMIC $\mu$ -KERNELS

Memory allocation for the spawn memory space occurs before threads are scheduled onto the SMs. The size of this memory depends on two parameters. The first parameter is the total number of threads able to be launched given the resource requirements. The second parameter is the amount of memory required to be passed between threads. Off-line, the compiler computes these two parameters, then uses them to determine the size of the spawn memory. When the sum of the spawn memory and shared memory (also computed off-line) exceeds the amount of on-chip memory, the hardware allocates the spawn memory to off-chip memory.

An assembly template for a  $\mu$ -kernel is shown in Example 2. Software instructions store and load the thread states. Before spawning a new thread, threads must save active data to memory (lines 14 and 15). Each thread has access to its spawn memory space through the *spawnMemAddr* special register created at thread launch (line 3). After the software instructions save the state of the thread to memory, the program calls the spawn instruction (line 17 or 19). The spawn instruction requires two arguments, a program counter for the  $\mu$ -kernel to be executed and the spawn memory pointer. While the parent thread is free to continue executing application code, the thread must not modify the spawn memory space, as the usage of the spawn memory by the child thread cannot be determined by the parent



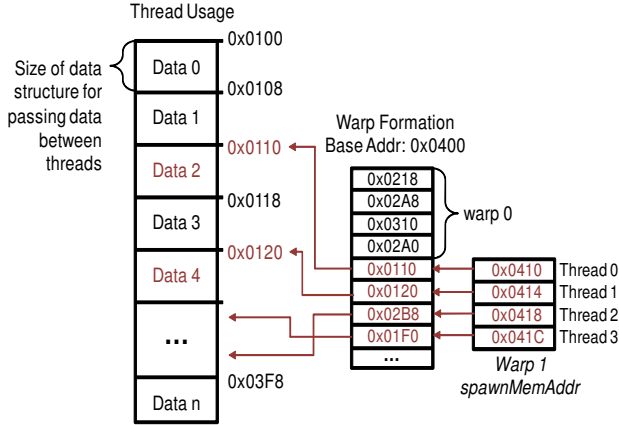


Figure 6. Spawn memory layout for threads accessing parent thread data using 4 threads per warp and 8 bytes of storage between threads. Child threads use their special register to access the warp formation data. The warp formation data is a memory pointer to the parent thread data.

thread and modifications by the parent thread could result in concurrency errors. To load the state for a dynamically created thread, the spawn memory address must be retrieved.

Figure 6 shows the memory layout used by threads to access spawn memory data. Dynamic threads are provided a memory address in *spawnMemAddr*. This address points to the warp formation section of the spawn memory space that was being written to when the thread was first created. The value stored in the warp formation memory is the memory pointer used in the spawn instruction that created the thread and points to the thread usage data (lines 3 through 5 in Example 2). Once the spawn memory address is retrieved, the state can be loaded into the thread’s registers (lines 7 and 8). The contents of the spawn memory space is accessible for the lifetime of the thread. The spawn memory space is reused to spawn a new thread by writing data back into the memory space for the child thread (lines 14 and 15).

Requiring the current state of a thread to be saved to memory results in overhead for performing  $\mu$ -kernel execution. Performing multiple load/store operations at the beginning and end of each  $\mu$ -kernel results in additional instruction execution and increased instruction latency. The number of operations required is application-specific and determined by the amount of memory required to save the state and the memory data layout. Care must be taken in determining the location in the application for creating dynamic threads, since the overhead may be more than the branch instruction.

Converting current SIMT rendering algorithms to use  $\mu$ -kernels requires identifying critical branch statements that decrease processor efficiency more than the overhead for creating a dynamic thread. Branching statements can be either looping conditions or conditional branching statements that are typically data dependent and affect the runtime of the thread. In our ray tracing example, the three looping

### Example 2 Sample $\mu$ -Kernel Assembly Code

```

1: microKernel_label:
2: # Get memory pointer to threads spawn memory
3: mov    rd1, SREG.spawnMemAddr;
4: ld.spawnMem r1, [rd1+0];
5: mov    rd1, r1;
6: # Load thread state from spawn memory
7: ld.spawnMem f1, [rd1+0];
8: ld.spawnMem f2, [rd1+4];
9:
10: # Run micro-kernel code
11: # Sets P0 to select between two micro-kernels
12:
13: # Save thread state back to spawn memory
14: st.spawnMem [rd1+0], f1;
15: st.spawnMem [rd1+4], f2;
16: # Create a new thread
17: @p0 spawn $microKernel_option_1, rd1;
18: @p0 exit;
19: spawn $microKernel_option_2, rd1;
20: exit;

```

operations have a dramatic effect on the runtime for each thread and allows one long running thread to have a significant negative impact on the processor utilization.

## VI. EXPERIMENTAL SETUP

We modified the GPGPU-SIM [16] simulator for evaluating the performance and accuracy of creating threads during runtime for  $\mu$ -kernels. Our spawn instruction and spawn memory space were added to the simulator and configured to use on-chip SM memory. The simulator was configured to resemble an NVIDIA Quadro FX5800 GPU [17] with additional on-chip memory for the spawn memory space. Table I shows the specific configurations setup of our simulator.

We performed simulations using two different thread scheduling models. The *block scheduling* configuration represents the FX5800 thread scheduling hardware. Warps are only scheduled via block scheduling if there are enough SM resources for the entire thread block, where thread blocks are composed of multiple programmer-defined warps. The block scheduling method allows for synchronization between all threads inside of a block. The *thread scheduling* configuration ignores block resources and schedules as many warps to an SM as other resources allow. Dynamic thread creation is designed for *thread scheduling* since thread synchronization is not required and allows for improved hardware usage.

### A. Benchmark Kernels

We used two benchmark CUDA kernels for our experimentation. The control algorithm is a ray tracing CUDA application called Radius-CUDA [18], which we used for performance measurements for the PDOM and MIMD configurations. Radius-CUDA uses a *kd-tree* [13] acceleration structure and Wald’s ray-triangle intersection algorithm [19].



Resource	Traditional	$\mu$ -kernel	$\mu$ -kernel Minimum
Registers	22	20	16
Shared Memory	60 bytes	56 bytes	32 Bytes
Global Memory	388 bytes	384 bytes	0 Bytes
Constant Memory	128 bytes	24 bytes	8 Bytes
Spawn Memory	0	48 bytes	48 bytes

Table II  
KERNEL PROCESSOR RESOURCE REQUIREMENTS PER THREAD.

The second benchmark kernel implements dynamic  $\mu$ -kernels. The Radius-CUDA program was modified by removing the three looping operations and adding in state load/store operations and thread spawning. Modifying the Radius-CUDA application for dynamic  $\mu$ -kernels is currently done at the Parallel Thread Execution (PTX) assembly language [20] level. To generate initial PTX assembly code and to determine the resources required for each  $\mu$ -kernel, the original kernel written in CUDA C is broken up into multiple global function calls that are compiled separately. Individual global PTX functions are then manually combined into one large application containing all  $\mu$ -kernels. For compilation of C code using NVIDIA’s NVCC compiler [9], function arguments were used to emulate the spawnMemAddr special register. During manual instrumentation of the PTX assembly code, this memory operation is modified to instead use spawnMemAddr as a register. The implemented algorithm used by the  $\mu$ -kernels is the same for both benchmarks.

The per-thread kernel parameters are shown in Table II. By using dynamic  $\mu$ -kernels, the resources required per thread are less than the original kernel. This is a result of the  $\mu$ -kernels using fewer instructions and the use of spawn memory space for additional register storage. Using  $\mu$ -kernels results in 800 threads per SM. To achieve optimal performance for the traditional algorithm implementation using block scheduling, the block size is set to two warps, resulting in 64 threads per block and 512 threads per SM. The number of thread blocks that traditional hardware can run is then the limiting factor for the total number of threads that can run on a SM. While increasing the number of warps per block allows for a higher number of threads per SM, the additional un-coalesced memory operations resulting from high thread divergence degrades the performance below the performance of 64 threads per block.

384 bytes of the per-thread global memory is for maintaining a stack data structure used for traversing the kd-tree. The value is arbitrarily chosen, since different tree structures represent different scenes which result in different stack memory sizes. The value is large enough for the largest scene in our benchmark dataset, however, the entire memory space is not used and usage varies between scenes.

The memory required for storing the state and for creating each dynamic thread uses the same data structure. This data structure requires 48 bytes of data to be stored and three

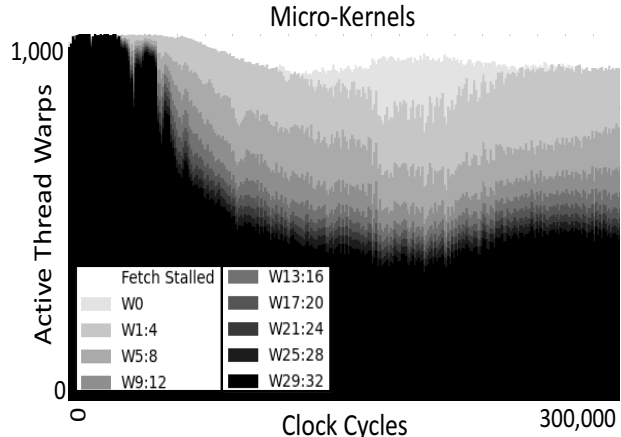


Figure 7. Divergence breakdown for warps using  $\mu$ -kernels for the conference benchmark. Warps are able to keep more threads active by creating new warps at critical branching points.

4-wide vector memory instructions are required for storing or restoring the state. The total memory then required for storing all possible thread states is less than the shared memory size. We implemented a naïve thread spawning method, where the entire store and restore operations for spawning a thread are performed for every loop iteration. Performance could be improved by first checking if a branch would cause divergence. If not, the branch could be taken by the warp instead of spawning new threads.

### B. Benchmark Scenes

Three benchmark scenes were used for testing the performance and efficiency of our dynamic  $\mu$ -kernels concept. Table III shows the rendered image and scene properties for each benchmark. *Fairyforest* tests ray traversal efficiency for large open spaces with areas of highly dense object count. *Atrium* contains a uniform distribution of highly dense objects through the entire scene. The *conference* benchmark has a high number of objects that are not evenly distributed throughout the scene.

## VII. EXPERIMENTAL RESULTS

To reduce simulation time, only the first 300k cycles were simulated at a resolution of 256x256. Simulation past 300k clock cycles results in similar behavior to the 150k to 300k range. The first 300k cycles were simulated to demonstrate the sharp drop in processor efficiency at the start of the application. Performance numbers for MIMD and PDOM were generated by running the original Radius-CUDA application on our simulator.

We start our analysis with the assumption of no bank conflicts for the spawn memory space. This assumption allows for simulation of future programming models or compiler optimization designed to eliminate a majority of the bank conflicts. Figure 7 shows the number of threads


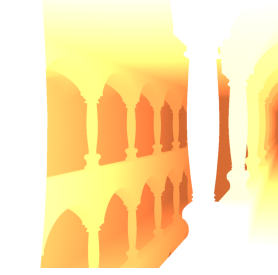
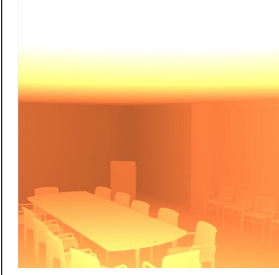
			
Benchmark	fairyforest	atrium	conference
Triangles	172,561	559,992	987,552
kd-tree Depth	36	37	35

Table III  
BENCHMARK SCENES WITH OBJECT COUNT AND TREE DATA STRUCTURE PARAMETERS.

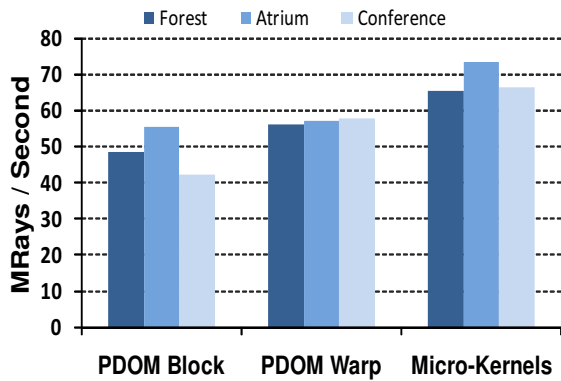


Figure 8. Performance results for all benchmarks using different branching and scheduling methods.

participating in all warps over time for the `conference` benchmark scene. The other two benchmarks have comparable plots. Similar to Figure 3, this plot categorizes a warp into 10 different categories based on the number of threads in a warp that are not idle. The SIMT processor efficiency can be directly correlated to this plot. Using dynamic thread creation allows for a much higher utilization of all the processors on a chip. The average instructions per cycle for the `conference` benchmark is 615,  $1.9\times$  higher than current hardware’s 326 instructions per cycle. Using dynamic thread creation still results in some processors running idle due to branching within spawned functions and pipeline stalls from memory operations.

Since dynamic thread creation introduces additional instructions, the instructions per cycle performance does not directly correlate to the rendering performance. Figure 8 shows the rate at which rays are processed for all benchmark scenes and both scheduling models. Block scheduling requires enough resources for the entire thread block before all threads in the block are executed. Warp scheduling will execute the maximum number of warps the hardware can support, breaking up blocks if an entire block does not fit. PDOM Warp achieves a higher performance than traditional hardware (PDOM Block) by allowing more threads to be

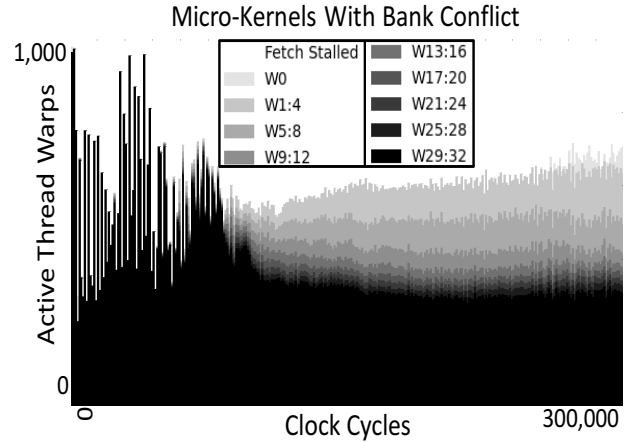


Figure 9. Divergence breakdown for warps using dynamic thread creation with bank conflicts for the `conference` benchmark. Warps still maintain more active threads over traditional branching methods, however, additional pipeline stalls are introduced by bank conflicts.

scheduled to an SM which allows for more memory latency to be hidden. Our dynamic threads are able to achieve higher performance, due to the reduction of critical branching statements and improved memory coalescing.

The creation of dynamic threads requires additional memory operations. Table IV shows the required bandwidth per image required to render a scene. Bandwidth values were computed from the number of down traversals and intersection tests required to render a single frame. The values are computed without any caching or separation between off-chip and on-chip memory spaces. The memory bandwidth required for dynamic thread creation is the difference between the traditional and dynamic values. The resulting overhead from creating dynamic threads results in an average bandwidth increase of  $4.4\times$  for reading data. The total increase in bandwidth for reading and writing is an average of  $7.3\times$ .

Bank conflicts do affect our performance results by introducing additional memory latency. Figure 9 shows the active threads within a warp simulated with bank conflicts

Benchmark	Reading	Writing	Total
fairyforest Traditional	62.1 MB	0.25 MB	62.35 MB
fairyforest Dynamic	296.5 MB	203.7 MB	500.2 MB
atrium Traditional	88.5 MB	0.25 MB	88.75 MB
atrium Dynamic	372.9 MB	258.1 MB	631.0 MB
conference Traditional	64.2 MB	0.25 MB	64.45 MB
conference Dynamic	263.3 MB	179.6 MB	442.9 MB

Table IV

MEMORY BANDWIDTH REQUIREMENTS FOR DRAWING A SINGLE IMAGE WITHOUT CACHING. VALUES ARE CALCULATED FROM THE NUMBER OF TREE TRAVERSAL OPERATIONS AND INTERSECTION TESTS.

for the spawned memory space. An increase in pipeline stalls is introduced with bank conflicts due to serialization of all conflicting bank memory operations to the spawn memory space. While the number of pipeline stalls has increased, processor efficiency is still superior to traditional branching methods and maintains an average instructions per cycle of 429, 1.3 $\times$  higher than current hardware.

Branching performance is shown in Figure 10 for the conference benchmark scene. Ideal memory systems were simulated (no memory latency) to determine the algorithm’s theoretical performance. PDOM has no performance increase when simulated with an ideal memory system, indicating its performance is limited by the branching hardware. Dynamic  $\mu$ -kernel execution improves performance up to 45% of the MIMD Theoretical with the potential to achieve 60% of the MIMD Theoretical.

## VIII. RELATED WORK

Improving wide SIMT processor efficiency for complex control flow has been approached from both hardware and software perspectives. Persistent threads [5] is a software scheduling algorithm specifically for ray tracing applications. This approach uses just enough threads to keep the machine full, and allows warps to focus on a single section of the entire algorithm. The entire algorithm is represented by multiple warps and uses memory arrays called work queues to pass data between warps. Warps are then executed in a loop operation for reading work from a queue, performing computations, and then writing back to another queue or device memory. Once the work queues are all empty, the threads then exit. Divergent control flow is reduced by allowing threads in a warp to write to different work queues. Since this is entirely a software solution, this algorithm can be applied to current and future hardware that supports the required memory transaction instructions. To prevent concurrency errors during work queue translations from the large number of active parallel threads, atomic instructions are required. Atomic instructions result in higher instruction latencies to serialize the instructions operating on the same data. Scheduling of warps is also left up to the developer, resulting in complex scheduling code for workload balancing, or simple methods that can result in an unbalanced distribution.

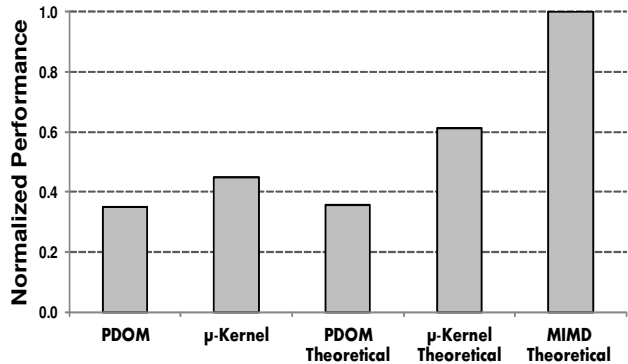


Figure 10. Branching performance for the conference benchmark. Theoretical results were simulated with an ideal memory system.

Hardware support for SIMT branching has grown in complexity as machines continue to advance to support wider application scopes. Early SIMT machines supported branching using mode bits [21] (also called predicated masks). Mode bits would disable the results for specific threads from being written back, effectively disabling the processor. This method results in every instruction being executed and processors are turned off accordingly. Modern day processors still implement mode bits for short branching instructions; however they do not support diverging control flow.

Dynamic warp formation [11] allows for warps to be modified to contain different threads during runtime. Processor cores have multiple warps assigned at application launch and only a few can be executed through the pipeline at a given time. As warps exit the pipeline back into the warp pool, threads are separated based on their next PC and placed into new warps. Warp metadata is then expanded to keep track of which threads are in a warp, so that register translation methods can still function correctly. Threads can be organized into warps using two different methods. The first requires that threads cannot change the SP that they have originally been assigned to. This method needs minimal hardware support, but limits thread flexibility. The second method adds a cross-bar network to all SPs to allow register values to be passed. While this adds hardware complexity, threads can be assigned to any warp that has a thread opening. Modifying warps at this level has the advantage of not requiring any code modifications.

Programming models for GPUs have focused on graphics rendering [22] and general-purpose computations [9], [23]. Graphics rendering programming models use the concept of a pipeline. While different pipeline stages can be programmable using custom kernels, additional pipeline stages cannot be added and the data movement between stages is fixed. Furthermore, implementation of the fixed logic hardware components is proprietary and closed. Programmable graphics pipelines is supported in GRAMPS [24], where

user-defined pipeline stages (implemented using kernels) pass data between stages using queues. However, in [24], underlying optimizations for wide SIMT control flow are not addressed.

## IX. CONCLUSIONS

In this paper, we proposed runtime thread creation to execute  $\mu$ -kernels in order to improve processor efficiency for global rendering algorithms. In our scheme, new threads are created to replace branching statements that cause low processor efficiency. We have presented our hardware architecture for creating new threads and grouping similar control flow threads into new warps. Our scheduler allows for scheduling of new warps when enough processor resources are available and can also schedule initial thread launches at the warp level instead of at a thread block level. By replacing critical branch statements with dynamic thread creation, we are able to increase the performance of rays per second by an average of  $1.4x$ .

Our original algorithm implementation is naïve in the sense that every loop is performed by spawning a new thread. Development of a more advanced algorithm can improve performance by allowing branching instead of thread creation when all threads in a warp follow the same branch. Additional future work is aimed at expanding supported algorithms beyond rendering algorithms and to support thread block level restrictions, such as thread synchronization and shared memory. By supporting a much larger set of applications, determining critical branches may not be as obvious as in the presented graphics rendering example. We plan to study the effects of selecting different code generation methods and to create a compiler to ease implementation from original code sources. While dynamic thread creation has shown to be advantageous for performance, the  $\mu$ -kernel concept can also benefit the SIMT programming model. To support such a model, we plan to further investigate the memory system for allocating memory to support an arbitrary number of thread creations per thread as well as caching architectures.

## ACKNOWLEDGMENT

This work has been supported in part by a National Science Foundation (NSF) Graduate Research Fellowship.

## REFERENCES

- [1] M. Hill and M. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, Jul. 2008.
- [2] M. Pharr and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., 2004.
- [3] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich, "Ray tracing on the Cell processor," in *IEEE Symposium on Interactive Ray Tracing*, Sep. 2006, pp. 15–23.
- [4] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, "Ray tracing on programmable graphics hardware," in *ACM SIG-GRAPH Courses*, 2005.
- [5] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on GPUs," in *Proceedings of High-Performance Graphics*, 2009, pp. 145–149.
- [6] D. Cederman and P. Tsigas, "On dynamic load balancing on graphics processors," in *Proceedings of Graphics Hardware*, 2008, pp. 57–64.
- [7] S. Rixner, W. Dally, U. Kapasi, B. Khailany, A. Lopez-Lagunas, P. Mattson, and J. Owens, "A bandwidth-efficient architecture for media processing," in *Proceedings of the Int'l Symposium on Microarchitecture (MICRO)*, 1998, pp. 3–13.
- [8] U. Kapasi, S. Rixner, W. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. Owens, "Programmable stream processors," *Computer*, vol. 36, no. 8, pp. 54–62, 2003.
- [9] NVIDIA, "NVIDIA CUDA Compute Unified Device Architecture Programming Guide," 2010.
- [10] M. Harman and S. Danicic, "A new algorithm for slicing unstructured programs," *Journal of Software Maintenance*, vol. 10, no. 6, pp. 415–441, 1998.
- [11] W. Fung, I. Sham, G. Yuan, and T. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *Proceedings of the Int'l Symposium on Microarchitecture (MICRO)*, 2007, pp. 407–420.
- [12] T. Whitted, "An improved illumination model for shaded display," *Communications of the ACM*, vol. 23, no. 6, pp. 343–349, 1980.
- [13] J. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.
- [14] P. Shirley and R. Morley, *Realistic Ray Tracing*. AK Peters, Ltd., 2003.
- [15] A. Ariel, W. Fung, A. Turner, and T. Aamodt, "Visualizing complex dynamics in many-core accelerator architectures," in *Proceedings of the Int'l Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2010, pp. 164–174.
- [16] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proceedings of the Int'l Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009, pp. 163–174.
- [17] NVIDIA, "NVIDIA Quadro FX 5800," [http://www.nvidia.com/object/product\\_quadro\\_fx\\_5800\\_us.html](http://www.nvidia.com/object/product_quadro_fx_5800_us.html).
- [18] Benjamin Segovia, "Radius-CUDA," 2008, <http://www710.univ-lyon1.fr/~bsegovia>.
- [19] I. Wald, "Realtime Ray Tracing and Interactive Global Illumination," Ph.D. dissertation, Computer Graphics Group, Saarland University, 2004.
- [20] NVIDIA, "NVIDIA Compute PTX: Parallel Thread Execution ISA 1.1," 2007.
- [21] W. Bouknight, S. Denenberg, D. McIntyre, J. Randall, A. Sameh, and D. Slotnick, "The Illiac IV system," *Proceedings of the IEEE*, vol. 60, no. 4, pp. 369–388, Apr. 1972.
- [22] M. Segal and K. Akeley, "The OpenGL Graphics System: A Specification (Version 4.0)," 2010.
- [23] ATI, "ATI Stream Computing, Compute Abstraction Layer Programming Guide 2.0," 2010.
- [24] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley, and P. Hanrahan, "GRAMPS: A programming model for graphics pipelines," *ACM Transactions on Graphics*, vol. 28, no. 1, 2009.