Control Flow Emulation on Tiled SIMD Architectures

Ghulam Lashari, Ondřej Lhoták, and Michael McCool

D. R. Cheriton School of Computer Science, University of Waterloo

Abstract. Heterogeneous multi-core and streaming architectures such as the GPU, Cell, ClearSpeed, and Imagine processors have better power/ performance ratios and memory bandwidth than traditional architectures. These types of processors are increasingly being used to accelerate compute-intensive applications. Their performance advantage is achieved by using multiple SIMD processor cores but limiting the complexity of each core, and by combining this with a simplified memory system. In particular, these processors generally avoid the use of cache coherency protocols and may even omit general-purpose caches, opting for restricted caches or explictly managed local memory.
We show how control flow can be emulated on such tiled SIMD architectures and how memory access can be organized to avoid the need for a general-purpose cache and to tolerate long memory latencies. Our technique uses streaming execution and multipass partitioning. Our prototype targets GPUs. On GPUs the memory system is deeply pipelined and caches for read and write are not coherent, so reads and writes may not use the same memory locations simultaneously. This requires the use of double-buffered streaming. We emulate general control flow in a way that is transparent to the programmer and include specific optimizations in our approach that can deal with double-buffering.

1 Introduction

GPUs are high-performance processors originally designed for graphics acceleration. However, they are programmable and capable of accelerating a variety of demanding floating-point applications. They can often achieve performance that is more than an order of magnitude faster than corresponding CPU implementations [1]. Application areas for which implementations have been performed include ray tracing, image and signal processing, computational geometry, financial option pricing, sequence alignment, protein folding, database search, and many other problems in scientific computation including solving differential equations and optimization problems.

These processors are best suited to massively parallel problems, and internally make extensive use of SIMD (single instruction, multiple data) parallelism. These processors do have multiple cores with separate threads of control, but each core uses SIMD execution. We will refer to such an architecture as a *tiled SIMD architecture*. Although we will focus on the GPU in this paper, even on the Cell processor and other general-purpose multi-core processors, higher performance can be achieved by a tiled SIMD approach to vectorization.

The simplicity of the tiled SIMD architecture enables a number of optimizations that result in higher performance. The techniques we will present can also apply to pure SIMD machines, such as the ClearSpeed processor. A tiled SIMD architecture can be emulated on a pure SIMD machine, simply by executing the per-tile computations serially instead of in parallel, just as concurrent threads can be emulated on a serial machine.

A major difficulty with pure SIMD machines is the efficient implementation of control flow. In a pure SIMD machine, control flow can be naively emulated by taking all paths of execution and then conditionally discarding results. However, this approach can result in arbitrarily poor efficiency. We show how to automatically and efficiently emulate general control flow on a tiled SIMD architecture.

We use GPUs as our test platform. This means that we have to deal with several other issues that are specific to GPUs, in particular double-buffering. However, our general approach can apply to any other tiled SIMD architecture.

We assume the processors are programmed using a stream processing model [1–3]. The model is based on applying a simple program, called a kernel, to each element of an input array. Conceptually all invocations of the kernel execute in parallel on each element. Kernel invocations cannot retain state or communicate with each other during computation. The order of execution of kernel invocations is also unspecified. In practice, for execution on a tiled SIMD machine, the arrays are strip-mined and the kernels are executed over small subsets (tiles) of the input arrays, using a combination of parallel and serial execution.

In the SPMD (single program, multiple data) variant of the stream processing model, the kernels can contain control flow; in the SIMD variant, they cannot. Our goal is to transform an SPMD kernel into a schedule of SIMD kernels over tiles, so that SPMD kernels can be executed on (tiled) SIMD machines.

We assume SIMD kernel execution is guarded; a guard is a predicate which controls whether the kernel will execute based on a data-dependent condition. If the guard fails for all the elements in a tile then computation on that tile is aborted, *actually avoiding work.* Aborted tiles do not write any output. However, if only some guards fail then the computation proceeds in SIMD fashion over the tile, but the outputs are masked: no output is written for elements for which the guard failed. We also assume that the hardware provides a count of the guards that did not fail across the entire stream after a kernel is executed over that stream. We will call this the completion count (CC). Finally, we assume the hardware uses deep memory pipelining and that reads and writes to the same memory location are not permitted during a single kernel invocation. This requires double-buffering of inputs and outputs.

We prototyped our multi-pass partitioning (MPP) technique on a GPU as GPUs have all the above-mentioned features; the guard can be implemented with a discard or early z-cull operation and the completion count with an occlusion count. In the conclusions we will discuss two modifications that could be made to tiled SIMD architectures to simplify control flow emulation.

Our contribution is a technique to efficiently emulate the SPMD stream processing model on a tiled SIMD machine. Our technique automatically partitions SPMD kernels with arbitrarily complex control flow into a set of SIMD sub-kernels without any internal control flow, only guards. We show how to automatically generate and run an efficient data-dependent schedule of these SIMD sub-kernels.

2 Previous Work

Our work is related to previous work on partitioning large computations with unbounded resource usage into multiple passes that respect hardware resource limits, and to prior approaches for the control flow emulation on SIMD machines.

Several algorithms have been suggested to virtualize limited hardware resources by partitioning GPU programs. However, existing algorithms partition only straight-line programs comprised of a single basic block. Chan [4] and Foley [5] devised the RDS (Recursive Dominator Split) and MRDS algorithms, respectively, that greedily merge operations in the dependence DAG into passes. Riffel [6] redefined the partitioning problem as a job shop scheduling problem and devised a greedy list scheduling algorithm called MIO. Heirich [7] presented an optimal algorithm based on dynamic programming for the MPP problem. However, none of these algorithms handle input programs with data dependent loops. Therefore, they do not work in the presence of control flow.

If-conversion is a technique that has been widely used in the context of vector machines to transform control dependence into data dependence. Both branches of the if statement are executed, but the writes are selectively performed, for components of the vector operand, to preserve the original results of the if statement. Our technique is more general than if-conversion in that it handles arbitrary control constructs including loops.

Purcell [8] manually partitioned a ray tracing algorithm composed of three nested loops into a set of conditionally executed blocks so it could run on a GPU that had no native control flow. The ray tracer was split into four kernels: ray generation, grid traversal, intersection, and shading. The execution of kernels on rays was controlled by a state variable that was maintained for each ray. The states corresponded to the kernels; for example, the traversing kernel would only run on those rays that were in the *traversing* state. Purcell then created a static schedule of the kernels that included running the ray generation kernel once, then repeatedly running traversal and intersection kernels, and then invoking the shading kernel once.

We generalize Purcell's approach and automate the partitioning so it can work on any control flow graph. Also, in the implementations we have studied, Purcell terminated the computation based on a user-provided number of iterations. Our approach instead automatically terminates the computation when all the stream elements have terminated. We also automatically handle the double buffering required by the fact that inputs cannot be bound as outputs in a single pass due to read-write hazards. We will compare our results directly to Purcell's ray-tracing implementation to evaluate performance but our techniques apply to any algorithm that requires control flow.

The idea that an arbitrary control flow graph can be implemented by predicating each basic block and wrapping all the basic blocks inside a single iteration is quite old (at least forty years [9]). Harel calls it a “folk theorem” and surveys many proofs [10]. This approach to implementing control flow has also been used to show that goto is unnecessary, since in fact all programs can be implemented using a “flow-chart simulator” that uses only structured control flow [11].

An alternative way to efficiently execute conditionals on SIMD machines is to use conditional streams [12, 13]. A single input stream is routed into multiple output streams and a separate kernel is then applied to each output stream. Similarly multiple input streams can be merged into a single output stream and a single kernel applied to the output stream. This however requires special hardware for stream compaction and expansion and also reorders the stream elements. Our mechanism does not reorder the stream and requires much simpler hardware support. Our technique is based on guarded kernels and tiling of the input stream. If the individual tiles are coherent, this will result in avoiding work for an entire tile for which the guard fails.

It should be noted here that the most recent generation of GPUs do support internal control flow on tiles; they do this by maintaining a hardware stack of conditional masks and temporary values for each SIMD tile. However, our approach permits simpler hardware and can potentially be used on pure SIMD machines (such as the ClearSpeed processor) as well. Even if control flow is implemented using a hardware mechanism, some resource may be limited, such as loop nesting depth, number of registers, or amount of local memory. In this case kernel partitioning will be needed. If control flow is present the kernel will have to be partitioned using techniques similar to those presented here even if the control flow is not itself directly limited.

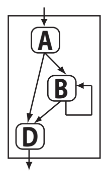
3 Control Flow Emulation on SIMD Machines

Our approach consists of three components: program partitioning, handling of state retained between partitions, and dynamic scheduling.

3.1 Program Partitioning

Our technique is capable of handling programs with arbitrary control flow constructs with arbitrary nesting depth and iteration counts. We first create a control flow graph of such programs that is then passed as input to our partitioning algorithm.

The basic idea behind our partitioning algorithm is to split the control flow graph of the program on the basic block level. A kernel is created for each basic block. For each element of the input array, we maintain (in a separate array) a program counter that indicates the next basic block to execute. Each kernel is predicated to run the basic block code only if the program counter matches



(a)

```
i = i + float2(1,1);
if (i.x > 20) break;
```

(b)

```
void main (in float2 coords : TEXCOORD0,
           uniform sampler2D STATE_IN,
           uniform sampler2D I_IN,
           out float2 I_OUT : COLOR0,
           out float2 STATE_OUT : COLOR1)
{
  float2 state = tex2D(texState, coords);
  // 1 is the identifier for block B
  if (state.x != 1)  discard;
  float2 i = tex2D(I_IN, coords);
  i = i + float2(1,1);
  I_OUT = i;
  STATE_OUT.x = i.x > 20 ? 2 : 1; // 2 = D
}
```

(c)

```
void main (in float2 coords : TEXCOORD0,
           uniform sampler2D STATE_IN,
           uniform sampler2D I_IN,
           uniform float LTIME,
           uniform float CTIME,
           out float2 I_OUT : COLOR0,
           out float2 STATE_OUT : COLOR1)
{
  float2 state = tex2D(texState, coords);
  //  1 is the identifier for block B
  if (state.x != 1 || state.y <= LTIME)  discard;
  float2 i = tex2D(I_IN, coords);
  i = i + float2(1,1);
  I_OUT = i;
  STATE_OUT = float2((i.x > 20 ? 2 : 1), CTIME);
}
```

(d)

Fig. 1. (a): A simple GPU program with a loop. (b): Basic block B of the program. (c): Kernel for basic block B that discards an array element if the state (i.e. program counter) does not match with the basic block identifier. (d): Kernel for basic block B that uses timestamped program counter.

the basic block number; otherwise, the basic block code is skipped. To mimic the execution of the original program, we run a schedule of these kernels. The schedule is maintained on the host CPU that repeatedly takes a kernel off the schedule and invokes it on the SIMD device. In later sections, we will show how an efficient schedule can be derived.

Figure 1(a) shows the control flow graph of a simple program we will use as an example. Figure 1(b) shows the code of basic block B from the original program, and Figure 1(c) shows the corresponding kernel for that block after the program has been partitioned.

Application of the original program to an array results in individual activations of the program taking different paths in the graph. Each activation may also perform a different number of loop iterations. In contrast, after partitioning we schedule individual basic block kernels in some particular order, running some of them repeatedly. Predication is used to ensure that execution of a given kernel processes only those array elements on which the code should execute according to the original program. We implemented predicates using the GPUs' conditional *discard* feature, although it might also be possible to use the early z-cull feature or a GPU if statement.

In order to ensure that the transformed code has the same semantics as the original program, we must capture the control flow decisions of individual array elements as they flow along the control flow edges. We do so using "program counters" (maintained in the SIMD device memory). We assign a unique identifier to each basic block. Before the schedule is run, the program counters for all the array elements are initialized to the identifier of the entry node in the control flow graph. Upon invocation of a kernel, only array elements whose predicates are true execute the corresponding basic block. The predicate inside the kernel ensures the program counter of the array element matches the identifier of the corresponding basic block. At the end of the basic block execution, the program counters are updated to reflect which basic block to execute next, possibly based on data-dependent information specific to each array element being processed. If the program counter of an array element does not match the currently executing kernel, the array element is not processed, and its program counter remains unchanged; the array element will be processed the next time the kernel matching its program counter is executed. The exit nodes in the control flow graph set the program counter to a unique identifier ∞ indicating that the array element has completed. Figure 1(c) shows Cg code that implements the predicate and updates the program counters to capture the control flow decisions.

The target hardware cannot read and write from the same array in one pass, but program counter arrays must be read, modified, and written in each kernel. Therefore, we must allocate two arrays α and β to double-buffer program counters. At each kernel invocation, one of these arrays is bound as input and the other as output. As a kernel finishes execution, the newly computed program counters for the executed array elements are written to α or β, whichever is bound as output to the kernel.

3.2 Temporary Variables

When the program is split into separate kernels, the values of temporary variables must be communicated from one kernel to the next. Like program counters, these temporary variables reside in the SIMD device memory across all passes in order to avoid data transfer overhead to and from the host CPU.

We identify three kinds of temporary variables in the original program: *BB variables* are those that are never live across a basic block boundary, *exposed variables* are those that are live across a basic block boundary but are never read and then written in the same block, and *RW variables* are those that are live across a basic block boundary and are read and then written in the same basic block. BB variables need no special treatment after program partitioning. Exposed and RW variables, however, must be saved into array memory after they are defined and restored before they are used. Exposed variables do not require double-buffering, so a single array suffices for each exposed variable. For each RW variable, we allocate two arrays for double-buffering. Each exposed or RW variable is turned into an array input, output, or both for kernels in which it is read or written.

We now show how these kinds of variables are identified. For each basic block in the original program, we compute two sets of variables: upward exposed *uses* (UEU) and downward exposed *defs* (DED) [14]. These variables become the array inputs and outputs, respectively, of the corresponding kernel. The UEUs for each basic block are those variables that are live at the beginning of the block; they are computed using a local liveness analysis. The DEDs for each basic block are variables that are written inside the block and live at the end of the block, as computed by a global liveness analysis of the whole program. Each variable that is both a UEU and a DED for the same block is a RW variable; every other variable that is a UEU or a DED for some block is an exposed variable.

Simply turning all variables that are live at the beginning and end of a basic block into inputs and outputs, respectively, would give correct semantics, but would require reading and writing every variable that is live across a basic block, even if it is not used within the block. Using UEUs and DEDs instead avoids this inefficiency.

Unlike program counters, RW variables need not be double-buffered in all kernels, but only in those in which they are read and then written. We will discuss the binding of RW variable arrays to kernels in Section 3.4 and 4.2. The Cg code in Figure 1(c) shows how to save and restore the RW variable i.

3.3 Dynamic Scheduling Using a Worklist

To decide which kernels to execute and with which program counter bindings, the host CPU maintains a dynamic worklist that is updated based on the completion counts. Each element of the worklist is a kernel with either a left or right arrow. For example, $\underrightarrow{B}$ indicates that kernel B should be executed with α as input and β as output, while $\underleftarrow{B}$ indicates the opposite bindings. The worklist is initialized with the initial basic block $\underrightarrow{A}$. In each pass, a kernel is taken off the worklist and executed. If the resulting completion count is not zero (i.e. the guard succeeded for at least one array element), all immediate static successors (as given by the control flow graph) are added to the worklist, with a program counter binding that is the opposite of the current pass.

The worklist ensures that a kernel X will not execute unless one of its predecessors was run since the last run of X. In our running example, A will run only once, since it has no predecessors.

If kernels X and Y need to be executed and there is a control flow path from X to Y but not from Y to X, X should be executed before Y, since executing X will require executing Y after it. To ensure this, of all the kernels appearing on the worklist, we always execute the kernel that appears earliest in a reverse post-order traversal of the control flow graph. This order is precomputed before execution begins. In our running example, this order delays execution of D until all iterations of B have completed, so that D is only run once on all array elements together.

3.4 Stale State

When an array element is discarded, no output is written for it. Therefore, if kernels discard array elements that are skipped, program counters for skipped elements cannot be updated in the output array. After executing a kernel that inputs program counter array α and outputs program counter array β, up-to-date program counters will be in array β for those elements that were executed, and in array α for those elements that were skipped. Neither array will have up-to-date program counters for every array element.

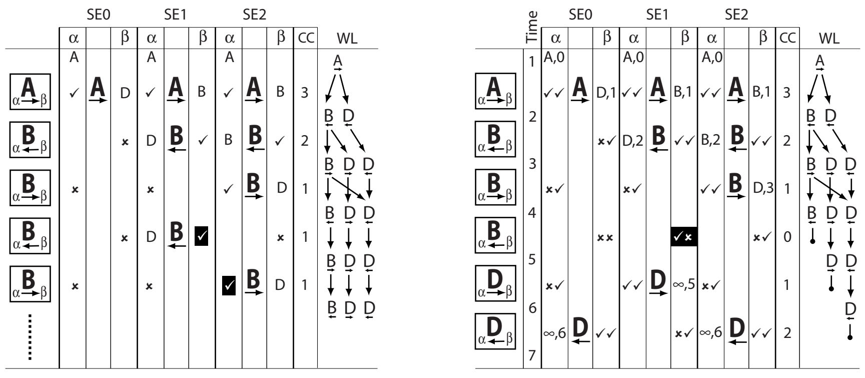


Fig. 2. (a): A stale program counter causes extraneous execution of block B–resulting in non-termination. (b): Timestamp identifies a stale program counter and prevents extraneous execution of block B

Stale program counters may cause extraneous execution of basic blocks and even prevent the computation from terminating in some cases. Figure 2(a) illustrates this with an example of the execution of the program from Figure 1. The control flow graph consists of three basic blocks A, B, and D, where B is a self-loop. The program is run on three array elements SE0, SE1, and SE2, making zero, one, and two loop iterations, respectively. The first column shows the kernel executed in each pass. The arrow under the kernel name indicates whether the program counter is read from array α and written to β, or vice versa. The column labeled CC shows the completion count for each kernel executed. The last column shows the contents of the worklist, in reverse post order, between passes. The columns in the middle show the progress of each of the array elements. Each of them is further subdivided into three sub-columns. Each middle sub-column shows the basic blocks that execute. The left and right sub-columns show the values of the two program counter arrays α and β. A ✓ indicates that the current program counter matches the kernel being executed, so the basic block appears in the middle sub-column, and the new program counter appears in the opposite program counter array. A $\times$ indicates that the current program counter does not match the kernel, so the basic block is not executed, and the opposite program counter array is not updated.

In the first pass, all three array elements execute block A, and the new program counters (D or B) are written to β. Kernels $\underleftarrow{B}$ and $\underleftarrow{D}$ are added to the worklist. In the second pass, SE0 discards while SE1 and SE2 execute B and kernels $\underrightarrow{B}$ and $\underrightarrow{D}$ are added to the worklist. In the third pass, SE2 completes the second iteration of B, adding $\underleftarrow{B}$ to the worklist while both SE0 and SE2 discard. In the fourth pass, the stale program counter B is found in β for SE1, causing B to be executed again, even though it was already executed in the second pass. This is an extra execution of B that was not specified by the original program. In addition, it results in addition of $\underrightarrow{B}$ to the worklist, triggering another execution of B. Finally, because β is not updated, the same thing will continue to happen in every second future pass, so the kernels will loop indefinitely.

In order to prevent the above problems caused by stale program counters, we add timestamps. With each kernel execution, we increment a global clock (maintained on the host CPU). We maintain a timestamp (in array memory) for each program counter, and two timestamps (in CPU memory) for each kernel, one for each program counter array binding (i.e. $\underrightarrow{A}$ and $\underleftarrow{A}$). Each program counter is stamped when it is generated, and a kernel is stamped when it is executed with the corresponding binding. The key idea is that a program counter A in α is stale if and only if it was generated earlier than the most recent execution of kernel $\underrightarrow{A}$ (since that execution will have generated a more up-to-date program counter in β). Thus, to avoid extraneous execution due to stale program counters, we need only modify the predicate of each kernel to skip the basic block if the program counter is stale, even if the program counter matches the basic block.

The code applying timestamps to our running example is given in Figure 1(d). Figure 2(b) shows the execution of schedule with timestamped program counters. The global clock appears in the column labeled Time. The first three passes of the schedule are the same as in Figure 2(a). In fourth pass, however, SE1 is discarded in kernel $\underleftarrow{B}$ because the timestamp of the program counter in β (1) is lower than the most recent execution time of $\underleftarrow{B}$ (2). Thus, the extraneous execution is avoided, and computation terminates in six passes as the worklist becomes empty.

Timestamps prevent the extraneous execution and non-termination caused by stale program counters. However, the schedule is still inefficient: we must execute block D twice. We will show an improvement in the next section.

RW temporary variables suffer from the staleness problem as well, as they are double-buffered. We solve the RW variable staleness problem by keeping the RW variable array bindings synchronized with the program counter array bindings (i.e. always copy all RW variables in every kernel). Since the timestamps prevent execution when the program counter is stale, and double-buffering of variables is synchronized with double-buffering of the program counter, the same timestamps also prevent use of stale variables. However, copying all variables in every kernel is inefficient; we will describe a better approach in the following section.

4 Optimizations

Two important optimizations can significantly reduce extraneous execution of kernel partitions and data copying: graph bipartization and node bypassing.

4.1 Graph Bipartization

When a block is reached with two different program counter array bindings, it must be executed twice, once for each binding. Moreover, the resulting program counters will again be split between both arrays, so blocks that come after it must also be executed twice. We would like to transform the control flow graph to avoid this double execution of blocks.

Specifically, we must transform the graph to be 2-colourable (equivalently, bipartite or containing no odd cycles). At run time, a kernel must be executed with a given binding if one of its predecessors has been executed with the opposite binding. If it is possible to statically assign bindings to the basic blocks such that the binding for a block is the opposite of the bindings of all its predecessors, then at run time, each kernel only ever needs to be executed with its statically assigned binding. Since the bindings can be thought of as colours for nodes, this is possible exactly when the graph is 2-colourable.

We make the graph 2-colourable by inserting additional *copying nodes* on existing edges of the graph. Each copying node simply copies program counters from its input array to its output array. Although copying takes some time, the cost of running a copying node is likely to be cheaper than having to run a possibly large basic block twice, along with all blocks executed after it. Still, we want to minimize the number of copying nodes.

To determine where to place copying nodes, we must find the smallest subset of edges that must be removed from the control flow graph in order to make it 2-colourable. This problem, Bipartite Subgraph, is a dual of Max-Cut and is NP-complete [15]. Each of the removed edges must join vertices of the same colour in the resulting 2-colouring; otherwise, it would not have to be removed, contradicting the optimality of the subset. Therefore, placing a copying node on each edge that is to be removed is equivalent to removing it. The minimal set of edges on which copying nodes must be placed is exactly the solution of Bipartite Subgraph.

To make bipartization computationally tractable we use an approximation algorithm for Max-Cut. Fortunately, a well-known approximation algorithm is available that runs in linear time, is easy to implement, and achieves a $\frac{1}{2}$-approximation [16].

Once the graph has been made 2-colourable, colours can be assigned in a simple traversal of the graph. Figure 3 shows a version of our running example made 2-colourable by adding copying nodes C_1 and C_2, and the corresponding multipass execution of the basic blocks. Block D is now executed only once, and the number of executions for block B have reduced from three to two.

4.2 Node Bypassing

As explained in Section 3.2, temporary variables are divided into BB variables, exposed variables, and RW variables. Only RW variables are double-buffered. Just like program counters, the values of RW variables could be stale. As described in Section 3.4, a simple but inefficient way to avoid the problems caused by staleness is to keep the variable double-buffering synchronized with program counter double-buffering, so that the same timestamp can be used for both. Unfortunately, this requires extra data copying.

One way to avoid copying all RW variables in all kernels is to add a separate timestamp for each RW variable, to avoid the need to synchronize with program

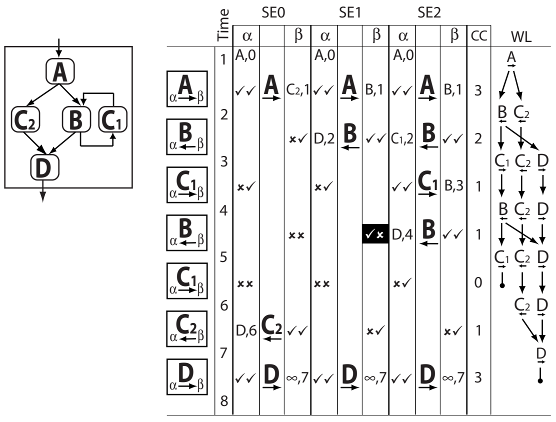


Fig. 3. Bipartization avoids double execution of block D.

counter double-buffering. However, if a cycle in the control flow graph contains an odd number of nodes that read and write the variable, the nodes in this cycle would have to be executed twice, once for each array binding for the variable. A program with n double-buffered variables could require 2^n combinations of variable array bindings. Requiring a basic block to be executed 2^n times is not practical.

We propose a more efficient solution that reduces the wasted memory bandwidth without requiring additional timestamps. For each variable, we define *RW blocks* as those that contain a read followed by a write of the variable, and *free blocks* as those that do not. The problem is then to find a subset of free blocks in which to copy the variable so that in every cycle, the total number of RW blocks and copies is even. This property, which we call the *Node Bypass Property*, makes it possible to statically assign variable array bindings to each basic block. Since we have made the control flow graph bipartite, we know that the set of all free blocks is one solution. To improve efficiency, we would like to minimize the number of copies (i.e. maximize the number of bypassed copy nodes). We have shown this problem to be NP-complete (by reducing Max-Cut to it). In the opposite direction, we have also reduced it to Weighted Max-Cut, so heuristics for Weighted Max-Cut can be used to find approximate solutions to Max Node Bypass. Sketches of both constructions appear in the appendix.

5 Experimental Results

We implemented our partitioning technique in the OpenGL backend of the Sh code generator [17]. Sh is a shader metaprogramming language and a precursor to the RapidMind development platform [18]. The hardware used for the experiment consisted of a 3.0GHz Intel Pentium4 with 1.0GB of RAM and an NVIDIA GeForce 8800 GTS GPU with 320MB of memory. The software was tested under

Scenes	Tris	Voxs	TriList Size	Iter (HP)	Iter (AP)	Time (HP)	Time (AP4)	Time (AP8)	Time (HP/AP4)	Time (HP/AP8)
Glassner04	840	140	1618	339	681	5222	5847	4311	89%	121%
Glassner05	840	315	2142	279	563	4196	4842	3580	87%	117%
Glassner10	840	2340	3886	111	227	1647	2096	1545	81%	107%

Table 1. For each scene, the number of triangles and voxels and the size of the triangle list is shown. Also the number of iterations of the ray tracing loop required by the hand-partitioned (HP) and auto-partitioned (AP) ray tracer program is shown. The last five columns show absolute and relative execution times of the three versions of the ray tracer: hand-partitioned using 4 outputs (HP), auto-partitioned using only 4 outputs (AP4), and auto-partitioned using all 8 outputs (AP8).

the Windows XP operating system with NVIDIA video driver version 97.73 and Cg compiler version 1.5.

We used a ray tracer with a number of scenes to evaluate the quality of partitions and schedule generated by our partitioning technique. We took the hand-partitioned ray tracer implementation bundled with the distribution of BrookGPU [19] and ported it to Sh. We also rewrote the ray tracer as a single GPU program with nested loops. The rewritten ray tracer was then passed as input to our partitioning algorithm. For comparison, our algorithm then automatically partitioned the ray tracer and executed the resulting kernels using dynamic scheduling. We compared the performance of the hand-partitioned ray tracer against that of our automatically partitioned implementation (see Table 1).

The hand-partitioned ray tracer consists of seven kernels: Three of them generate eye rays and static and dynamic data for traversal, the fourth traverses the voxels, the fifth computes ray-triangle intersections, the sixth validates the intersections, and the seventh performs shading for the intersection points. The traversal, intersection, and intersection validation kernels run repeatedly using a host CPU loop. Unlike the automatically partitioned version, the hand-partitioned version provides no automated way to determine when computation has completed. We determined the minimum number of iterations required for each scene manually by comparing the output of runs with different numbers of iterations. The render times shown in Table 1 for the hand-partitioned ray tracer are for the minimum number of iterations required for each scene.

The control flow graph of the ray tracer originally contained 12 basic blocks. To give each kernel a reasonable amount of computation, we automatically merged basic blocks that were split only due to *if* statements (not loops) using conditional assignments. This reduced the number of basic blocks to 5.

Our partitioning algorithm automatically partitioned the ray tracer into five kernels. The algorithm employed all the techniques presented in Sections 3.3, 3.4, 4.1, and 4.2. The Graph Bipartization algorithm introduced two copying nodes, increasing the number of kernels to seven. The liveness analysis revealed eleven temporary variables that were live across basic block boundaries in the control flow graph. Seven of them were exposed variables while four were RW variables that required double-buffering. The max-node-bypass approximation algorithm then detected that all but one node for each of these four variables could be bypassed. Hence only one extra kernel was needed to read and write these RW variables in order to keep them synchronized with the program counters.

Note that automatic partitioning can take advantage of any hardware improvements, but hand-partitioned code must be rewritten to do so. For example, the hand-partitioned ray tracer was optimized for an older GPU, so it utilized only four of the eight available outputs of the improved hardware. Our automatically partitioned ray tracer utilized all eight, saving us from further partitioning the kernels to virtualize the hardware's output resources. We also evaluated a version of the auto-partitioning algorithm limited to using four outputs for a fair comparison with the hand-partitioned algorithm.

We ran the hand-partitioned and the automatically partitioned ray tracers on three different scenes that consisted of different number of voxels and triangles. The details about the scenes are given in Table 1. When allowed to use all eight outputs, our auto-partitioned ray tracer was 7% to 21% faster than the hand-partitioned one. However, even when artificially limited to four outputs, our auto-partitioned ray tracer was only 11% to 19% slower than the hand-partitioned version. In general the auto-partitioned ray tracer also performed more repetitions than the hand-partitioned implementation. We attribute this to the generality of our solution. The hand-partitioned ray tracer was written by experts in the ray tracing area who were able to structure the ray tracing kernels so that more work could be done in fewer executions of the kernels, but our auto-partitioning algorithm has little knowledge about the algorithm implemented in the program being partitioned. Moreover, some overhead is due to the occlusion query that determines when execution has terminated; in the hand-partitioned implementation, the number of iterations to execute must be determined manually. Finally, use of the occlusion query requires adding timestamps to handle stale program counters, which uses additional memory bandwidth.

6 Conclusions and Future Work

This paper presented a mechanism to emulate SPMD stream processing behaviour on a tiled SIMD machine using the GPU as a test platform. We showed how to automatically partition programs containing arbitrary control flow and to schedule the resulting partitions. We showed how to automatically detect the termination of computation using a completion count mechanism (implemented using an occlusion count feature on the GPU) and presented solutions to stale state problems that arose from double-buffering. The solutions to stale state included timestamps, graph bipartization, and a node-bypass optimization. A worklist-based algorithm was also presented for dynamically scheduling the partitions. A ray tracer performed 7% to 21% faster when partitioned using our automatic technique than a hand-tuned manually partitioned version, because our automatic technique took advantage of all available hardware resources, while the manually partitioned version was optimized for an older GPU. Even when artificially limited to using the same resources as the manually partitioned version, the automatically partitioned ray tracer was only 11% to 19% slower—although it was also more general in that it was automatically checking for completion.

We realized during this work that the complications arising from stale state could be avoided if the hardware implementing the guard allowed writing to the outputs before the guard was evaluated. Unfortunately, on current GPUs, the discard operation masks out all outputs, including writes that occurred before the discard. Thus, when the guard fails, we cannot copy state from the input arrays to the output arrays, resulting in stale state. Avoiding stale state by using a different strategy for predication might eliminate the need for timestamps, graph bipartization, and solving the max node bypassing problem.

Another way to avoid stale state would be to remove the hardware restriction that prevents both reading and writing the same memory location in a kernel. If double buffering was not required, we could keep the state of the stream elements in one set of arrays, again avoiding stale state. In fact, either one of these modifications, on its own, would suffice to eliminate stale state and its associated complication.

In the future, we would like to explore additional optimization techniques. A better scheduling technique that prioritized the kernels for scheduling based on the amount of work accumulated for them could deliver better performance. The use of the completion count could also be limited to only some passes instead of all of them. A major performance improvement can be achieved by packing the intermediate results into different components of the same tuple of an array, but this is not easy to implement because of the double-buffering. We statically inserted copying nodes to avoid running certain kernels twice, but dynamically inserting the copying nodes, when needed, could result in less copying. Moreover, one should probably insist on adding copying nodes that will do the least amount of work as opposed to the minimum number of nodes. For example, if a path on the control graph is taken 90% of the time and the other is taken 10% of the time, it is better to insert two nodes on the latter rather than one on the former.

As current GPUs and other high-throughput targets are actually SPMD machines and can handle limited control flow, it would be useful to generalize our approach and use it as part of a resource virtualization framework. In the general context it is important to merge control flow and generate larger SPMD kernels to avoid the overhead of invoking a large number of small kernels. Additional transformations should be explored to restructure the program so that it fits within hardware limits, while maximizing efficiency and advantage of the limited control flow provided by the GPU.

References

1. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Lefohn, J.K.A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. In: Eurographics '05: State of the Art Reports. (2005) 21–51
2. Das, A., Dally, W.J., Mattson, P.: Compiling for stream processing. In: PACT 2006: Parallel Architectures and Compilation Techniques, ACM (2006) 33–42
3. McCool, M.D.: Scalable Programming Models for Massively Multi-Core Processors. In: Proc. IEEE. (Jan 2008)
4. Chan, E., Ng, R., Sen, P., Proudfoot, K., Hanrahan, P.: Efficient partitioning of fragment shaders for multipass rendering on programmable graphics hardware. In: HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware. (2002) 69–78
5. Foley, T., Houston, M., Hanrahan, P.: Efficient partitioning of fragment shaders for multiple-output hardware. In: HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware. (2004) 45–53
6. Riffel, A., Lefohn, A.E., Vidimce, K., Leone, M., Owens, J.D.: Mio: fast multipass partitioning via priority-based instruction scheduling. In: HWWS '04: ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware. (2004) 35–44
7. Heirich, A.: Optimal automatic multi-pass shader partitioning by dynamic programming. In: HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware. (2005) 91–98
8. Purcell, T.J., Buck, I., Mark, W.R., Hanrahan, P.: Ray tracing on programmable graphics hardware. ACM Transactions on Graphics **21**(3) (July 2002) 703–712
9. Cooper, D.C.: Böhm and Jacopini's reduction of flow charts. Commun. ACM **10**(8) (1967) 463
10. Harel, D.: On folk theorems. Commun. ACM **23**(7) (1980) 379–389
11. Knuth, D.E.: Structured programming with *go to* statements. ACM Comput. Surv. **6**(4) (1974) 261–301
12. Kapasi, U.J., Dally, W.J., Rixner, S., Mattson, P.R., Owens, J.D., Khailany, B.: Efficient conditional operations for data-parallel architectures. In: 33rd Annual IEEE/ACM International Symposium on Microarchitecture. (2000) 159–170
13. Popa, T.S.: Compiling Data Dependent Control Flow on SIMD GPUs. Master's thesis, University of Waterloo (2004)
14. Marlowe, T.J., Ryder, B.G.: Properties of data flow frameworks: a unified model. Acta Inf. **28**(2) (1990) 121–163
15. Garey, M.R., Johnson, D.S.: Computers and Intractability. W. H. Freeman and Company, San Francisco (1979)
16. Sahni, S., Gonzalez, T.: P-complete approximation problems. Journal of the ACM **23**(3) (July 1976) 555–565
17. McCool, M.D., Qin, Z., Popa, T.S.: Shader Metaprogramming. In: Proc. Graphics Hardware. (September 2002) 57–68
18. McCool, M.D.: Data-Parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform. In: Proc. GSPx Multicore Applications Conference. (Oct–Nov 2006)
19. Buck, I.: BrookGPU. `http://graphics.stanford.edu/projects/brookgpu/` (2003)

A Complexity of Max Node Bypassing

Theorem 1 *The Max Node Bypassing Problem is NP-hard.*

Proof (Proof Sketch). We show a reduction from an instance of Max-Cut to an instance of Max Node Bypassing. Given an instance of Max-Cut $G\langle V, E\rangle$, create a control flow graph G' using the "gadgets" shown in Figure 4 as follows.

- *For each node $v \in V$, create a gadget A as shown in Figure 4 with both $\bigcirc$ and $\bigtriangledown$ labeled v.*
- *For each edge $v \rightarrow w \in E$, create a gadget B as shown in Figure 4 with both $\Diamond$ and $\Box$ labeled "$v \rightarrow w$", and connect the $\Box$ labeled "$v \rightarrow w$" to the $\bigcirc$ labeled w and the $\bigtriangledown$ labeled v to the $\Diamond$ labeled "$v \rightarrow w$".*

The resulting control flow graph G' is an instance of the Max Node Bypassing Problem with the $\Box$s as free nodes, and $\bigtriangledown$s, $\Diamond$s, $\bigcirc$s as RW nodes. G' is bipartite. It can be proved that a set $C \subseteq E$ is a cut if and only if the corresponding $\Box$s labeled with edges in C satisfy the Node Bypassing Property in G'. Therefore the maximal cut in G corresponds to the maximal node-bypass in G'.

To approximately solve an instance of the Max Node Bypassing Problem, we reduce in the **opposite** direction, from Max Node Bypassing Problem to Weighted Max-Cut. We can then solve the Weighted Max-Cut using known approximation algorithms, such as a variant of [16].

Given an instance of Max Node Bypassing, a bipartite control flow graph $G\langle V, E\rangle$ and set of free nodes $S \subseteq V$, we create an instance of Weighted Max-Cut $G'\langle V', E'\rangle$ as follows. For each node $v \in S$, we create gadget C shown in Figure 4. For each node $v \notin S$, we create gadget D. For each edge $v \rightarrow w \in E$, we merge $v_3 \in V'$ with $w_1 \in V'$. We assign weights to all the edges as follows: each dashed edge has weight 1, the weight of each solid edge is the total number of dashed edges plus 1. It can be shown that the set of solid edges in G' make a cut. The Max-Cut in G' must therefore include at least the solid edges and possibly some dotted edges. It can also be shown that a subset $\sigma \subset S$ of free nodes satisfy the Node Bypassing Property if and only if the set of dashed edges in gadgets C corresponding to nodes in σ, combined with the set of all solid edges, is a cut in G'. Therefore, the maximal subset satisfying the Node Bypassing Property corresponds to the Max-Cut in G'. Thus, an approximation to the Max-Cut in G' can be mapped back to an approximately maximal node bypass in G.

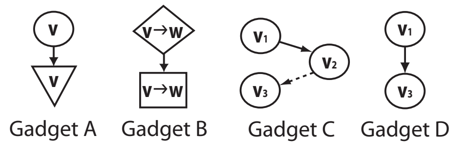


Fig. 4. Gadget graphs