MIMD Interpretation on a GPU

Henry G. Dietz and B. Dalton Young

University of Kentucky, Electrical and Computer Engineering

Abstract. Programming heterogeneous parallel computer systems is notoriously difficult, but MIMD models have proven to be portable across multi-core processors, clusters, and massively parallel systems. It would be highly desirable for GPUs (Graphics Processing Units) also to be able to leverage algorithms and programming tools designed for MIMD targets. Unfortunately, most GPU hardware implements a very restrictive multi-threaded SIMD-based execution model.
This paper presents a compiler, assembler, and interpreter system that allows a GPU to implement a richly featured MIMD execution model that supports shared-memory communication, recursion, etc. Through a variety of careful design choices and optimizations, reasonable efficiency is obtained on NVIDIA CUDA GPUs. The discussion covers both the methods used and the motivation in terms of the relevant aspects of GPU architecture.

1 Introduction

Both conventional wisdom and engineering practice hold that a massively parallel MIMD machine should be constructed using a large number of independent processors and an asynchronous interconnection network. However, it may be beneficial to implement a massively parallel MIMD using what is essentially SIMD hardware because those structures scale more efficiently to larger numbers of processing elements. For example, at this writing, both Intel and AMD are producing multi-core processor chips with no more than 8 MIMD processing elements – compare this to the hundreds of SIMD-based processing elements found on the latest Graphics Processing Unit (GPU) chips from NVIDIA [1] and ATI [2], or even to the 96 processing elements on ClearSpeed's more conventional SIMD chip [3]. The primary disadvantage in using a SIMD microengine for MIMD execution is that execution of different types of instructions must be serialized. Often, this implies that some fraction of the processing elements must be temporarily disabled awaiting their instruction's turn, but even having a very small fraction enabled can yield more active processing elements per chip than would be present in a fully-active MIMD chip of similar circuit complexity.

The concept of MIMD execution on SIMD hardware was at best a curiosity until the early 1990s. At that time, large-scale SIMD machines were widely available and, especially using architectural features of the MasPar MP1 [4], a number of researchers began to achieve reasonable efficiency. For example, Wilsey, et al., [5] implemented a MasPar MP1 interpreter for an instruction set

Algorithm 1 Basic MIMD Interpreter Algorithm

1. Each PE fetches an instruction into its instruction register (IR) and updates its program counter (PC).
2. Each PE decodes the instruction from its IR.
3. Repeat steps 3a-3c for each instruction type:
   (a) Disable all PEs where the IR holds an instruction of a different type.
   (b) Simulate execution of the instruction on the enabled PEs.
   (c) Enable all PEs.
4. Go to step 1.

called MINTABS. That instruction set is very small (only 8 instructions) and is far from complete in that there is no provision for communication between processors, but it did provide basic MIMD execution via interpretation. Our first MIMD interpreter running on the MasPar MP1 [6] achieved approximately 1/4 the theoretical peak native distributed-memory SIMD speed while supporting a full-featured shared-memory MIMD programming model.

The simplest method by which a MIMD instruction set can be implemented using SIMD-based hardware is to write a program that interpretively executes that MIMD instruction set. Such an interpreter has a data structure, replicated in each SIMD PE, that corresponds to the internal registers of each MIMD processor. Hence, the interpreter structure can be as simple as Algorithm 1. Unfortunately, there are several reasons why such an interpreter might not be efficient for GPU execution.

1.1 Interpretation Overhead

The most obvious problem is that interpretation implies some overhead for the interpreter; even MIMD hardware simulating a MIMD with a different instruction set would suffer this overhead. Although interpreter overhead varies widely, an order of magnitude slowdown is common.

Fundamentally, step 2 of Algorithm 1 involves the use of something like a C `switch` statement to decode each instruction. There are many ways to implement such a construct. Traditional SIMD machines often have control units that are capable of efficiently executing "mono" operations that effect all processing elements. Thus, such machines may be able to use highly efficient mechanisms like mono jump tables to decode instructions. The NVIDIA CUDA [1] documentation mentions such an indirect jump instruction, but it was never used by their compilation system in our testing. Worse still, CUDA apparently uses a very slow linear sequence of branches to implement `switch` – so optimized branch trees can be more appropriate.

In addition, fundamentally, SIMD hardware can only simulate execution of one instruction type at a time. This implies that the time to interpret an instruction for each processing element is proportional to the sum of the interpretation

times for each instruction type. Fortunately, there are tricks that can be used to minimize this effect.

1.2 Indirection

Still more insidious is the fact that even step 1 of Algorithm 1 cannot be executed in parallel across all PEs in many SIMD computers. The next instruction for each PE could be at any location in the PE's memory, and many SIMD machines do not allow multiple PEs to access different memory locations simultaneously. On such a machine, any parallel memory access made will take time proportional to either the number of different PE addresses being fetched from or the size of the address space which could be accessed. (For example, this is the case on the TMC CM1 and TMC CM2 [7].) Note that step 3b suffers the same difficulty if load or store operations must be performed. Because many operations are limited by memory access speed, inefficient handling of these memory operations easily can make MIMD interpretation infeasible.

This overhead can be averted only if the SIMD hardware can indirectly access memory using an address in a PE register. Examples of SIMD machines with such hardware include the PASM Prototype [8], MasPar MP1 [4], and the ClearSpeed CSX [3]. Fortunately, all modern GPUs have this ability, although random accesses can cause memory bank or cache conflicts. Better still, the texture memory access facilities can be used to speed-up and remove contention from indirect access to read-only objects. Textures are particularly appropriate for instruction fetches.

For current GPUs, objects that can be written – i.e., data objects – have banking constraints imposed on indirect access. Using ATI CAL [2], there is a strict "owner writes" policy. NVIDIA CUDA [1] allows random access, but cost increases by an order of magnitude for accesses that cannot be coalesced into access of a single line across memory banks. Originally, CUDA GPU coalescing worked only for access patterns in which each bank is accessed only by its owner; the latest version allows permutation of owners and banks. In either case, the result is that better performance can be achieved if the interpreter system is structured to increase the probability of conformant data access patterns.

One consequence of the banking constraints is a preferred data memory layout of 32-bit `datum_t` like:

```
datum_t mem[NPROC / WARPSIZE][MEMSIZE][WARPSIZE];
```

in which `NPROC` is the number of logical processing elements (assumed to be a multiple of `WARPSIZE`). One might think that a 2-dimensional layout with `mem[MEMSIZE][NPROC]` would suffice, but that would significantly complicate address arithmetic because `NPROC` is not necessarily a power of 2 and might not even be a compile-time constant. In fact, the CUDA compilation system does not handle the constant power of 2 stride of `WARPSIZE*sizeof(datum_t)` any better, but explicitly using shift and mask operations on pointer offsets implements the desired addressing without multiply and modulus operations.

1.3 Enable Masking

The above algorithm assumes that it is possible for PEs to enable and disable themselves (set their own masks). Although most SIMD computers have some ability to disable PEs, in many machines it is either difficult to have the PEs disable themselves (as opposed to having the control unit disable PEs, as in the PASM prototype [8]) or some arithmetic instructions cannot be disabled because they occur in a coprocessor, as in the TMC CM2 [7]. In such cases, masking can be circumvented by the use of bitwise logical operations, e.g. a C-like SIMD code segment:

```
where (ir == CMP) { cc = alu - mbr; }
```

might be implemented by all PEs simultaneously executing:

```
mask = -(ir == CMP); cc = ((cc & ~mask) | ((alu - mbr) & mask));
```

which is relatively expensive. This works because the C `==` operator always returns the integer value 0 or 1. Notice that, in addition to the bitwise operations, the above implementation requires a memory access (i.e., loading the value of `cc`) that would not be necessary for a machine that supports enable masking in hardware. Because masking is done for each simulated instruction, the masking cost effectively increases the basic interpretation overhead. Examples of SIMD machines whose hardware can implement the appropriate masking include the TMC CM1, the MasPar MP1 [4], and ClearSpeed CSX [3].

The SIMD variant execution model implemented by GPUs has significantly more complex support for masking, which offers the potential to dramatically improve performance. Instead of being a single, relatively wide, SIMD engine, modern GPUs actually consist of a group of smaller SIMD engines, each apparently processing what NVIDIA calls a "warp" at a time [1]. These narrow SIMD engines are augmented by scheduling hardware that implements multithreaded control. In effect, this type of GPU hardware provides three different types of hardware enable implementation.

Divergent Flow The relatively coarse-grain implementation of enable masking, the mechanism for handling divergent control flow (as implemented by NVIDIA CUDA [1]) essentially converts masking into a thread scheduling operation in which the hardware serializes execution of code for different enable sets. For example, the body of an `if-else` statement in which odd numbered processing elements take the `then` clause and even ones take the `else` clause would behave as though the current warp was divided into two separate warps, each of which has only half the processing elements enabled. At the point where control flow rejoins, these merge to create a fully enabled warp.

Predication Rather than treating a conditionally-executed block of instructions as a schedulable unit, predicated instructions can be used to execute the code. Predicated instructions are unconditionally executed, but side-effects are disabled for those processing elements in which the predicate is false. Side effects include not only writing of results, but also reading of operands. For example, a

false-predicate store to or load from global memory does not cause bank conflicts [1].

Skipping In the case that no processing element within a "warp" of SIMD threads is enabled, the scheduler simply will not schedule that warp. This is a fundamental difference between a many-small-SIMD-engines GPU and a conventional wide SIMD engine. A conventional SIMD machine with 16,384 PEs will tie-up all 16,384 PEs with code in which only PE 0 is enabled, but a GPU with a "warp size" (SIMD thread width) of 32 could merely block the resources of PE 0 through PE 31, with the scheduler skipping the code for all the other warps. This is a very powerful feature, because it implies that the lowest possible utilization is 1/32; for example, if there are 500 different instruction types to interpret, one would expect serialization over all 500 types – but the worst case would be over 32 types present within a particular warp.

The NVIDIA CUDA environment does not provide direct control over the choice between different methods for implementing branches or enable masking, instead automatically selecting based on its own cost estimate. However, we have implemented pragma that can be inserted (using inline assembly directives) and then used to control a conversion script we wrote to perform minor rewrites on PTX code before completing the compilation. For example, one pragma allows branches that are known to be non-divergent ("mono" branches) to be marked as such. However, we have not seen large performance improvements by using this simple PTX-rewriting mechanism.

The remainder of this paper discusses the overall design and implementation of interpreter-based MIMD On GPU (MOG). Section 2 discusses instruction set design for the interpreted MIMD system. In section 3, a brief overview of the programming language and compiler environment is given. Section 4 describes the assembler for the simulated instruction set. Several versions of the simulator, and performance on a variety of NVIDIA CUDA systems, are described in Section 5. Conclusions are presented in Section 6.

2 Instruction Set Design

One might expect that the best instruction set to use for the MOG system would be the native instruction set for the GPU, but that generally is not the case. The problem comes from the fact that two instructions with the same opcode, but referencing different registers, are not able to be factored across processing elements without increasing the decode overhead.

Because they lack explicit operands, stack instruction sets tend to factor better. However, implementing them is most efficient using indirect addressing, which register files typically do not support. Using NVIDIA CUDA [1], indirect addressing of a stack in global memory would be too expensive. Fortunately, there are alternatives – using a stack in registers or in block shared memory. The shared memory is not really shared, but provides the indirect addressing needed to allow factoring; the implementation using registers always keeps the

top of the stack in a specific register and shifts the others up or down as needed. Either way, the stack is too small, so the proposed model uses a split stack. The "block stack," or bstack, is the small fast stack, and objects are explicitly moved between it and the stack in global memory. Our system also provides the option of treating a small amount of shared memory or actual registers either as a register file or a tiny cache.

The remainder of the instruction set architecture is relatively straightforward, designed to minimize the number of different types of instructions both statically and dynamically while keeping instructions simple and efficient. For example, rather than having a relatively rare call instruction, the return address is explicitly `PUSH`ed on the stack and a `JUMP` instruction is used to invoke the routine. The instruction set also avoids having multiple data object sizes. Data memory is accessed as an array of 32-bit objects, which can be treated as either integers or floats. Smaller objects are extended and larger objects become tuples of 32-bit objects. Communication is done using global memory loads and stores.

System calls are implemented by the `SYS` instruction. Most `SYS` functions are actually implemented by performing a barrier synchronization across all `SYS` participants, which immediately terminates the current interpreter fragment and allows host code to handle the call. This mechanism also can be used to invoke native CUDA code fragments as functions from within an interpreted code.

3 Language and Compiler (`mimdc`)

Even though the target instruction set architecture is very simple, it was found to be nontrivial to re-target an existing compiler to generate MOG assembly code. The problem is that most compilers expect a general-register target, and the explicit caching of the stack in the bstack was just different enough from conventional stack manipulation to be awkward for re-targeting from other stack machine targets. For these reasons, a proof-of-concept compiler for a modest, but usable, C subset was created from scratch using the `PCCTS/Antlr` tools.

That C compiler is called `mimdc`. It is straightforward and does no flow analysis, but does perform a number of simple optimizations based on constructing and walking expression trees. In addition to the usual constant folding and simplifications, the compiler performs a number of simple rewrites aimed at increasing the frequency of specific instructions and idiomatic sequences. Fundamentally, these rewrites could be viewed as partial normalization of code evaluating expressions, with the goal of making source code that looks very different yield highly factorable machine code.

The language accepted by `mimdc` is a dialect of ANSI C slightly extending what we implemented for the MasPar MP1 [6]. The dialect supports recursive functions, pointers, single-dimensional arrays, the usual control-flow constructs, etc. It also provides a "parallel subscript" lvalue suffix operator for access to data structures in the memory of any processing element; `a[||b]` refers to the variable `a` on processing element `b`. In addition, the language accepted by the compiler includes a variety of intrinsic functions, which correspond to system

calls, so that these operations can be directly coded without the need for a C interface wrapper.

4 Assembler (`mogasm`)

It is expected that more complete C and FORTRAN compilers will be re-targeted to generate MOG assembly code in the near future, thus, having a separate assembler that can be used with any of the compilers is important. Further, the simulation system is constructed in such a way so that full MIMD – not just SPMD – code is permitted. In fact, it is possible to have a completely different program, even written in various different languages, for each individual processing element. This feature is implemented by allowing the output assembly code from each program's compilation to be merged by the MOG assembler, `mogasm`.

Aside from some conditional assembly support intended to facilitate smart merging of multiple assembly source files and assembly-coded library routines, `mogasm` is a fast, but very conventional, multi-pass assembler. The generated object code consists of a set of data structure definitions including segments for the program text (machine code) and initialized data; these structures are directly compiled into a CUDA program to implement the MIMD application as an ordinary host-executable file. All data is generated as 32-bit objects. However, the machine code representation is allowed to use 8, 16, or 32-bit objects and the opcodes and format easily can be changed. As discussed in Section 5.2, our optimization tools are capable of automatically redefining the instruction set encoding to minimize the decode overhead – literally creating a new coding scheme for each MIMD application, if desired. Although the opcode assignments vary, it appears that a 16-bit opcode with optional 16-bit arguments encoding generally works best for code. It takes too many 8-bit objects to encode a 32-bit constant and the NVIDIA CUDA system seems to generate poor code for some operations on 32-bit opcodes.

5 The Simulator (`mogsim`)

The MOG simulator, `mogsim`, is a surprisingly small program. None of the versions thus far has contained more than 2,500 lines of C and CUDA source code[1]. However, the program is not as simple as its small size suggests.

5.1 PE Environment

Figure 1 shows the logical structure of a processing element's environment for `mogsim`. The processing element (PE) itself is a virtual PE. The number of virtual PEs is not trivially derived from the number of physical PEs, but is

[1] Of course, that number does not count the compiler, assembler, and other helper programs, which embody over 70,000 lines of code.

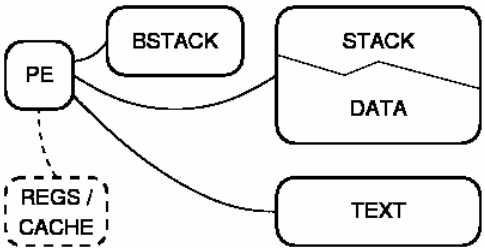

Fig. 1. MOG Processing Element Structure

computed as a function of the number of SIMD engines, the WARPSIZE, and various constraints that together determine the optimal degree of multithreading. Perhaps the most stringent constraint is that there are only 16K bytes of shared memory per SIMD engine and only 8K or 16K registers, which are divided by the multithreaded block size.

The bstack resides in what NVIDIA calls block shared memory, but this data structure is not actually shared. In fact, the bstacks of the various PEs within a block are deliberately skewed so that apparently consecutive objects on a PE's bstack actually have addresses that differ by a multiple of the warp size. In this way, there are no bank conflicts in referencing the bstack. Block shared memory is a very limited resource; 7 bstack entries seems to be a practical minimum, with some trade-off possible between a larger bstack and use of shared memory for registers and/or a tiny, software-implemented, direct-mapped cache.

In contrast, the stack and data are allowed to be quite large, because they are kept in an array in what NVIDIA calls global memory. The array is actually part of the same data structure for all PEs, with a 32-bit object `WARPSIZE`-strided layout (as described in Section 1.2) in order to ensure accesses to "local" PE memory are free of bank conflicts. The text segment is a texture map of 16-bit objects. All MIMD code for any PE is placed in the text segment with a simple contiguous layout. Not shown in Figure 1 are the PE internal registers including program counter (`pc`), stack pointer (`sp`), block stack pointer (`bsp`), and instruction register (`ir`).

5.2 Interpreter Structure

The `mogsim` interpreter is implemented on the GPU by a single function called `emulate()`. This function's fragment is not called just once, but repeatedly until the MIMD program has terminated, each time executing some fixed maximum number of interpreted instructions. This was done partly because GPUs impose a timeout in order to terminate errant fragments of code. However, the development machines are running an X windows display at the same time as the MIMD simulation, and system crashes and lock-ups were observed when codes were allowed to run for even a modest portion of the theoretically allowable maximum fragment time. Thus, we empirically determine a maximum safe interpret count

per invocation of `emulate()` for each target system and use a register variable called `serial` to count. Unless executing in trace mode, only a short vector of status information is transferred from the GPU to the host to determine if the MIMD code is requesting a system function when a fragment ends. Currently, the primary system function is `exit()`.

Within the interpreter, there are a number of tricks that could be used to improve execution speed. Some of these tricks apply only to certain types of interpreter structures, and there are a variety of approaches possible. Many variations were implemented and evaluated; only a few are reported here.

Sequence of Single-Instruction Subinterpreters (SIS) Nilsson and Tanaka [9] envisioned a set of subinterpreters such that each subinterpreter emulates only a single type of instruction and all subinterpreters are executed once per interpreter cycle. Before entering the loop over all subinterpreters, the first instruction is fetched. The main loop decodes each type of instruction in sequence and, if any PE wants to execute that instruction, the operation is performed and the next instruction is fetched. Decoding can be as simple as comparing the `ir` to the desired opcode. This type of interpreter is easily implemented, and has been used for Genetic Programming on a GPU [10].

However, using program coding and/or execution statistics, Nilsson and Tanaka [9] pick an order for the subinterpreters that is intended to maximize the expected number of instructions executed per processor per interpreter cycle. For example, given that the subinterpreters are in the order `PUSH`, `LD`, `ADD` then the instruction sequence `PUSH`, `LD`, `ADD` would take only one cycle – but `LD`, `PUSH`, `ADD` would take 2 cycles and `ADD`, `LD`, `PUSH` would take 3. In our earlier work targeting the MasPar MP1 [11], which used a fundamentally different approach, one of the techniques used was "frequency biasing" in which expensive operations were deliberately made to execute less frequently. Abu-ghazaleh et al. [12] later integrated this idea with the single-instruction subinterpreter sequencing concept by allowing subinterpreter sequences that do not incorporate all instructions in every interpreter cycle.

Our GPU implementation uses statistics collected from one or more traced MIMD application program runs to determine execution frequency of both individual instructions and *instruction digrams*. A fixed-length sequence of single-instruction subinterpreters is automatically constructed. The number of times the subinterpreter for a particular instruction occurs in that sequence is at least 1, but otherwise mimics the expected runtime frequency of occurrence of that individual instruction. A modified evolutionary search is then used to optimize the order of the subinterpreters so that the digram-frequency-weighted average span between subinterpreters for instructions appearing in each digram is minimized. For example, using 64 subinterpreters to cover the 37 instruction types, a random ordering usually has a weighted average span greater than 15 – meaning that at runtime an average of at least 15 subinterpreters would be attempted before executing a subinterpreter for the next instruction. The optimizer typically reduces that number to between 4 and 6.

It is useful to again note that this type of interpreter structure requires neither communication nor sophisticated control flow. Thus, with the potential change of implementing each subinterpreter (or short sequence of subinterpreters) as its own code fragment, this type of interpreter can be executed on *nearly any* GPU. For example, this could be implemented using ATI's Stream model [2], OpenCL [13], OpenGL [14] or DirectX [15] fragment code, etc.

Selection of a Present Instruction to Interpret (SIR) The shared memory implemented by CUDA GPUs suggests that it would be practical to interpret a single instruction each cycle which is selected from those instructions currently present in some PE's `ir`.

The first step is to fetch the current instruction for each PE into its instruction register, which is accomplished by a 1D texture fetch. In order to have all processing elements within a warp agree on the instruction to process this cycle, one processing element (thread) is selected from each warp and its instruction is used. In our system, `NPROC` is the total number of PEs and `IPROC` is the number of this PE in the range 0..(`NPROC`-1); `BNPROC` and `BIPROC` are the same concepts, but for execution within a block. Thus, by selecting the thread whose processing element number within its warp is equal to the interpreter cycle `serial` number modulus the warpsize, round-robin scheduling is implemented. The store into `*sharedsir` is a trivially conflict-free operation on block-shared memory because only one thread in each warp is enabled. Reading back the value into the `sir` register in each thread is essentially a broadcast within the warp, with implicit synchronization. The selected instruction type is executed by all PEs in which the `ir` held the same type of instruction. The result is that all PEs are ensured to make fair progress, and the decoding of the instruction in `sir` is essentially "mono" (control unit) code that can never diverge. This allows instruction decoding to be implemented by highly efficient O(1) methods, such as an indirect jump based on the opcode value.

The primary advantages for this approach are fairness and the fact that the cost associated with instructions that no PE currently wants to execute can be essentially zero, suffering neither decode nor enable masking overhead. Although Abu-ghazaleh et al. [12] and others argue that with large numbers of PEs every instruction type should be requested by some PE in every cycle, our earlier work using a 16,384 PE MasPar MP1 [6] found that many instruction types did *not* occur for many instruction cycles, and with a warpsize of 32 and 37 instruction types, it is literally impossible that all instruction types occur within a warp.

Unfortunately, as mentioned in Section 1.1, the CUDA environment implemented `switch` as a linear sequence of tests and predicated branches, making decode cost dependent on the ordering of cases and significantly higher than it logically should be. In order to improve decode speed, a helper program was built to generate a binary decode tree using `if` statements, but again the CUDA environment created inefficient code – in fact, there was virtually no timing difference between the non-divergent decode trees and a well-ordered `switch`. This

order dependence spawned the idea of making the decode tree have a variable depth, shallow for common operations, deeper for others.

The basic concept we use is essentially building an `if` decode tree with the same type of structure used for Huffman coding [16], although the frequency weightings in the interpreter decode tree are optionally adjusted to also prefer cheaper decoding for instructions that execute faster. Our system uses instruction frequency measurements (as discussed in Section 5.2) to automatically create the Huffman-like decode tree and to modify the assembler (see Section 4). The opcodes are remapped so the decode tree can use a simple "less than" comparison operation per level.

Divergent Factored Decoding (DFD) Ironically, although NVIDIA frequently warns of the performance hazards associated with divergent control flow, the overhead associated with it seems significantly less than the decode overhead of Section 5.2's SIR method without indirect jumps. Changing the above method to directly use the `ir` instead of `sir`, we create an interpreter structure that tends toward significant divergence, but partially factors the traversal of the decode `switch` or tree across instructions.

Subinstruction Factoring using CSI (CSI) The most effective method for the MasPar MP1 [4][6] factored-out portions of the process of interpreting an instruction across the various types of instructions present. For example, all binary stack operations fetch the top-of-stack (TOS) and next-on-stack (NOS) values, store a new result into the NOS, and then adjust the stack pointer. Thus, with an instruction set encoding designed to expose such commonality, it is possible to execute the bulk of the interpreter overhead for multiple types of instructions simultaneously, only masking the portions that distinguish the specific instructions.

In this algorithm, the code still begins by fetching the current instruction – and this is the only place in the interpreter loop in which the current instruction is fetched. The following steps do not execute an entire instruction at a time, but rather correspond to factoring-out the common parts of the "microcode" for each instruction and controlling execution of each "microinstruction" individually. More precisely, the factoring to which we refer involves application of an algorithm called Common Subexpression Induction (CSI) [17] to find and/or create the maximally beneficial set of common subexpressions among the microcode operations. For example, extracting an argument from the instruction stream can be factored-out for all instruction types that use an argument. Similarly, update of the block stack pointer can be done just once with the appropriate instruction-type-dependent adjustment.

This technique has the theoretical advantage that it maximally shares overhead for common microinstructions. One issue is that, because CSI tends to separate-out portions of an instruction, it naturally tends to increase the amount of state that needs to be held within the GPU. CSI execution also involves a lot of conditionals which, as we have discussed earlier, are not necessarily handled well

Table 1. Benchmark System Configurations

Feature	"Laptop"	"Desktop1"	"Desktop2"	"Desktop3"
Host Processor	Intel T8300	AMD 4200+	Intel 920	AMD 4200+
NVIDIA GPU (CC)	8600M GT (1.1)	8800 GTS (1.0)	9800 GT (1.1)	GTX 280 (1.3)
GFLOPS: Host/GPU	9.2 / 91.2	10.5 / 345.6	21.36 / 544.3	10.5 / 933
Power: Host/GPU	35 / 22	89 / 146	130 / 125	89 / 236
GPU Cores/PEs	32 / 1,024	96 / 2,304	112 / 3,584	240 / 10,560
Best Time: `perf`/`fact`	9.63 / 10.55	7.77 / 7.7	6.66 / 7.2	8.33 / 9.76

by the current NVIDIA CUDA environment. This approach also suffers from serialization of all the conditional tests, which is a more severe problem than it was for the SIR and DFD methods – because here there are more separate conditions to be tested. A large portion of the conditional tests can be factored using the ordering optimizations discussed for SIR (Section 5.2).

5.3 Performance

The performance evaluation reported in this paper centers on determining how effective each of the interpretation approaches is under various conditions and how large a performance penalty is incurred for support of a full-featured shared-memory MIMD programming model as compared to directly using the relatively restrictive SIMD-based native model for a GPU.

The most direct comparison possible would be performance of native CUDA code relative to that of a MIMD program, which implements precisely the same algorithm, with no program logic that would force different PEs to be executing different code simultaneously. For this purpose, we created a simple benchmark, `perf`, in which each virtual PE executes a tight loop containing 1,000,000 floating-point multiply-accumulate operations. It was specifically created to be a "worst case" CUDA-compatible benchmark for the MIMD environment – for example, `mogsim` does not implement multiply-accumulate as a single instruction as native CUDA does. A second benchmark code was created to determine the relative performance of each of the alternative interpretation methods when processing a MIMD program that makes extensive use of features CUDA does not support. This code, `fact`, involves each virtual PE performing 10,000 integer factorial operations using the usual recursive function. Inside of a loop, different PEs are deliberately given different factorials to compute, ranging from 0! to 21!, causing PEs to diverge quickly both across and within blocks and warps.

Performance Across Various GPUs One might expect a strong bias favoring, or even a requirement of, one of the higher-levels of "compute capability" (CC) among the machines listed in Table 1. However, none of the interpreters requires any features that did not exist in CC 1.0. Later versions do perform slightly better in that they relax the rules for coalescing memory references and may have more registers, but these differences are incremental for both `perf`

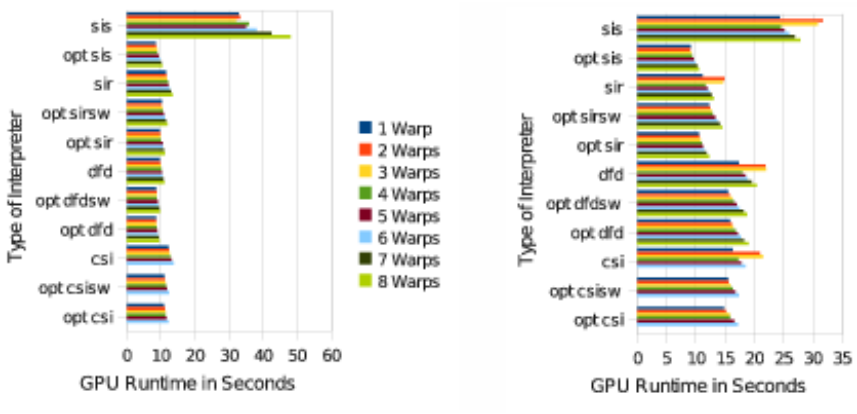

Fig. 2. Performance for Different Warp Counts (`perf` and `fact`) on "Laptop"

and `fact`. The primary improvement seen is that more work can be done in about the same time on systems that allow more virtual PEs. Executing the exact same program logic as `perf`, but coded as native CUDA code, takes 1.46 seconds using the "Laptop" system benchmarked above with 8 warps per SM (1,024 virtual PEs) – thus, our worst-case slowdown for the best interpretation method is only about 6.6X native. Using the `SYS` instruction mechanism within a MIMD interpreted program to call native CUDA code (as mentioned in Section 2) has an overhead of about 0.025 seconds per invocation. Thus, if one recognized that `perf` could be rewritten as native code and invoked it using the `SYS` mechanism from within a MIMD program, the slowdown in using the MIMD interpreter would be just 1.7%.

The recursive `fact` benchmark is effectively *infinitely faster* than native CUDA code, because CUDA supports neither recursion nor a sufficient number of simultaneous (or timesliced) instruction streams. One might think implementing only a few of the MIMD features on top of CUDA without an interpreter might yield much better performance, but just making native CUDA code "interuptable" at each global store resulted in a 4X slowdown [18] – `mogsim` is inherently interuptable at every instruction with no more than a 6.6X slowdown.

Performance of Different Interpretation Methods Figure 2 shows the results of trying all possible numbers of warps per SM for 11 different types of interpreters on the "Laptop" target system using as many blocks as there are SMs (in this case, 4). The basic methods correspond to those discussed in Sections 5.2 ("sis"), 5.2 ("sir"), 5.2 ("dfd"), and 5.2 ("csi"). The variants prefixed with "opt" use an order-optimized decode method instead of a fixed-depth decode tree – Huffman trees or ordered `switch` statements (if suffixed with "sw"). The number of warps was constrained primarily by the small number of registers available in the hardware, which hardware assigns in groups of 8; most of the interpreters used just 31 or 32 registers, except for variants using CSI, which

needed 34. Although the figure shows data only for the "Laptop" target, similar performance trends were seen on all the GPU targets.

The first observation one should make is that adding warps extends execution time very little while significantly increasing the work done; the highest performance always is achieved running the maximum number of warps per SM that the hardware will allow. For "Laptop" this means a total of 4*8*32 virtual processors, or 1,024 threads. It even seems that scheduler noise is higher for small numbers of warps, perhaps because the benchmark system was using the same GPU for computation and the primary X windows display. Other tasks may have cut-in, especially during some of the 2 and 3 warp runs of `fact`.

It also is clear that different interpreters handle SIMDish and highly divergent MIMD code with differing relative efficiencies. The "opt sis" method is the fastest for highly irregular MIMD code, whereas "opt dfd" is fastest for MIMD code with lower entropy. Although "sis" is terrible, the "opt sis" method does consistently well; it is only about 9% slower than "opt dfd" for the minimally-divergent code. The most consistently good performance is delivered by the "sir" variants, and "sir" itself is the overall best performer if it is required that the same instruction encoding and interpreter structure must be used for all MIMD programs. That said, "sir" was 24% slower than the best for `perf` and nearly 41% slower for `fact`. Interestingly, this version of "sir" is approximately an order of magnitude faster than the original version of "sir," which was the only MIMD interpreter demonstrated in our research exhibit at the SC08 conference last year.

6 Conclusion

In this first published paper on MIMD On GPU (MOG) interpretation, we have provided an overview of the motivation, general approach, and major implementation issues involved. Extensive optimization of various interpreter approaches was undertaken, incorporating a wide range of insights. The toolchain was designed to keep complexity out of the compiler; complex analysis and transformation all happens in the assembler and below, including automatic use of evolutionary computing to optimize the interpreter structure and instruction encoding based on data from test runs.

The results clearly show that MIMD interpretation is a viable method for high-performance computing, at least on a wide range of NVIDIA CUDA GPUs. Worst-case slowdown for code that could have been written in native CUDA is only 6.6X and, in such a case, the system would allow native code to be called as a function reducing overhead to less than 2%. Wildly dynamic MIMD code, including recursion, also works with apparently good efficiency.

The toolchain is expected to become a public domain source code release as soon as it has reached an appropriate level of usability and stability.

References

1. NVIDIA, "NVIDIA CUDA compute unified device architecture programming guide version 1.0," June 2007.

2. ATI, "ATI stream SDK user guide v1.3-beta," December 2008.
3. ClearSpeed, "ClearSpeed whitepaper: CSX processor architecture," *ClearSpeed Technology plc*, vol. PN-1110-0702, 2007.
4. T. Blank, "The maspar mp-1 architecture," *35th IEEE Computer Society International Conference (COMPCON)*, February 1990.
5. P. Wilsey, D. Hensgen, C. Slusher, N. Abu-Ghazaleh, and D. Hollinden, "Exploiting simd computers for mutant program execution," *Technical Report No. TR 133-11-91, Department of Electrical and Computer Engineering, University of Cincinnati, Cincinnati, Ohio*, November 1991.
6. H. G. Dietz and W. E. Cohen, "A massively parallel mimd implemented by SIMD hardware," *Purdue University School of Electrical Engineering Technical Report TR-EE 92-4*, p. 28 pages, January 1992.
7. Thinking Machines Corporation, "Connection machine model cm-2 technical summary, version 5.1," May 1989.
8. H. Siegel, W. Nation, and M. Allemang, "The organization of the PASM: Reconfigurable parallel processing system," in *Ohio State Parallel Computing Workshop*, pp. 1–12, March 1990.
9. M. Nilsson and H.Tanaka, "MIMD Execution by SIMD Computers," in *Journal of Information Processing, Information Processing Society of Japan, vol. 13, no. 1*, pp. 58–61, 1990.
10. W. B. Langdon and W. Banzhaf, "A SIMD interpreter for genetic programming on GPU graphics cards," in *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008* (M. O'Neill, L. Vanneschi, S. Gustafson, A. I. Esparcia Alcazar, I. De Falco, A. Della Cioppa, and E. Tarantino, eds.), vol. 4971 of *Lecture Notes in Computer Science*, (Naples), pp. 73–85, Springer, 26-28 Mar. 2008.
11. H. G. Dietz and W. E. Cohen, "A control-parallel programming model implemented on simd hardware," *Languages and Compilers for Parallel Computing, edited by U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Springer-Verlag, New York, New York*, 1993.
12. N. B. Abu-ghazaleh, P. A. Wilsey, X. Fan, and D. A. Hensgen, "Synthesizing variable instruction issue interpreters for implementing functional parallelism on SIMD computers," *IEEE Transactions on Parallel and Distributed Systems*, 1997.
13. Khronos OpenCL Working Group, "The OpenCL specification version 1.0," December 2008.
14. B. L. et. al., "Arb_fragment_program," *OpenGL Extension Registry*, vol. http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt, Aug. 2002.
15. Pixel Shader Reference, "http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/dx81_c/directx_cpp/graphics/reference/shader/pixel/pixel.asp."
16. D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.
17. H. G. Dietz, "Common subexpression induction," *1992 International Conference on Parallel Processing, Saint Charles, Illinois*, vol. II, August 1992.
18. Q. Hou, K. Zhou, and aining Guo, "Debugging gpu stream programs through automatic dataflow recording and visualization," May 2009.