Execution Of MIMD MIPSEL Assembly Programs Within CUDA/OpenCL GPUs

Henry G. Dietz and Frank Roberts
University of Kentucky, Department of Electrical and Computer Engineering

***Abstract*—Earlier work demonstrated that general MIMD-parallel programs could be transformed to be efficiently interpreted within a CUDA GPU. Unfortunately, the quirky split-stack instruction set used to make the GPU interpreter efficient meant that only a specially-constructed C-subset compiler could be used with the system.**

In this paper, a new system is described that can directly use a complete, production-quality, compiler toolchain such as GCC. The toolchain is used to compile a MIMD application program into a standard assembly language – currently, MIPSEL. This assembly code is then processed by a series of transformations that convert it into a new instruction set that manages GPU local memory as registers. From this code, an optimizing assembler generates a customized interpreter in either NVIDIA CUDA or portable OpenCL.

I. Introduction

Traditional SIMD architectures always have been very easy to scale, but clock rates generally plummet as fanout becomes large. GPUs (Graphics Processing Units) solve the classic SIMD clock rate problem by implementing a group of loosely-coupled relatively-narrow SIMD engines instead of a huge synchronous one. For example, ClearSpeed's more conventional CSX700 SIMD chip [1] reached a maximum clock rate of 250MHz, whereas contemporary NVIDIA [2] and AMD [3] GPUs ran at twice that rate. The potential efficiency of GPUs is further increased by avoiding hardware-intensive features that do not increase peak arithmetic performance. For example, features like interrupt handling, various forms of memory protection, and large caches would all reduce peak performance per unit circuit complexity.

The problem is that the resulting more scalable, but more complex and restrictive, execution model is difficult to program. The goal of the research reported here is making GPUs able to efficiently run parallel programs that were developed targeting MIMD systems using either shared memory or message-passing communication via MPI [4].

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 [5], a number of researchers began to achieve reasonable efficiency. For example, Wilsey, et al., [6] implemented a MasPar MP1 interpreter for a toy instruction set called MINTABS. Our first MIMD interpreter running on the MasPar MP1 [7] achieved approximately 1/4 the theoretical peak native distributed-memory SIMD speed while supporting a full-featured shared-memory MIMD programming model. Earlier versions of our MOG (MIMD On GPU) environment have demonstrated similar efficiency on NVIDIA CUDA GPUs using carefully tuned interpreters [8].

Given that the MOG concept has been proven viable, the work presented in this paper is more focused on making MOG practical. Beyond reimplementing and making minor improvements to the best methods discovered in earlier research, the primary contributions are:

- Rather than processing a stack assembly language, the new MOG system uses an accumulator/register instruction set. Both assembly languages needed somewhat unusual features in order to obtain good efficiency. However, management of the very limited low-latency memory resources is mapped into an apparently conventional register allocation problem in the new instruction set, rather than the unusual problem of explicit movement of data between local and global portions of a split stack. This makes the new instruction set much more compatible with existing compiler backends without imposing a significant runtime performance penalty.
- Whereas the old versions required compilers to be retargeted, the new version alternatively allows existing compiler toolchains to be used unchanged. The system described here converts MIPSEL assembly code into the new MOG assembly language, thus allowing any existing toolchain generating MIPSEL code to be used. The method has been tested with both LLVM [9] and GCC.
- Earlier versions of the MOG system exclusively targeted NVIDIA CUDA GPUs. The MOG system described in this paper targets both NVIDIA CUDA and portable OpenCL [10]. It worthwhile noting that although OpenCL is intended to be vendor neutral and is supported by both NVIDIA and AMD/ATI GPUs, writing code to be portable between GPUs from different vendors and efficient on all requires very careful use of OpenCL.

The new instruction set architecture most directly exposes the key issues, and is presented in Section II. Translation of MIPSEL assembly code is described in Section III. Section IV briefly discusses the MOG assembler and interpreter structure – more details about how and why MOG interpreters work can be found in our LCPC 2009 paper [8]. Conclusions are given in Section V.

II. Instruction Set Architecture

Earlier MOG systems were very closely tied to the properties of the NVIDIA CUDA GPUs they targeted. In contrast,

the new MOG system explicitly is designed to be able to be efficiently implemented by both NVIDIA CUDA GPUs and OpenCL on any of a wide variety of hardware including GPUs from both NVIDIA and AMD/ATI. Thus, it is useful to view the latest MOG as a true instruction set architecture offering many implementation choices. There is an abstract model of the hardware environment for each PE (processing element) and an instruction set specification including both assembly language and bit-level encoding. This is the compiler target for portable MOG.

A. PE Hardware Environment

Each PE as we count them is actually a virtual PE inside a GPU, generally not a dedicated block of physical hardware. The number of virtual PEs is not trivially derived from the number of physical PEs, but is computed as a function of the number of SIMD engines and various constraints that together determine the optimal degree of multithreading. The smallest GPUs contain at least 256 MOG PEs and the largest GPUs could contain more than 64K MOG PEs. A cluster of nodes each containing a GPU would multiply the PE count by the number of nodes, easily reaching millions of PEs.

For most purposes, each MOG PE behaves exactly like a conventional processor running a sequential user-level program. It apparently executes independently of both the other PEs and the host computer's processor. Although the architectural model could support fully general MIMD execution, the current compilation toolchain does require the text of the program to be a single image shared by all PEs: a MIMD variant more precisely known as SPMD (Single Program, Multiple Data). Each PE may apparently asynchronously take its own path through the program, but all PEs share the same program.

Whereas conventional processors handle system calls by entering a protected execution mode provided by the processor's hardware, MOG PEs have no such support. Indeed, most GPU hardware has no mechanism by which it can initiate anything like a system call, nor can it access I/O devices – other than the video display it drives. Our solution is to hand-off any such system call to be done by the host processor. A GPU PE cannot directly interrupt the host processor, but it can drop a message to the host in a place that both can access. The primary way a PE can attract the host processor's attention is to cleanly terminate the GPU kernel, thus causing the host to examine the message buffer and act as it directs. This is a very slow system call process, but fully general, and system calls are fairly slow even within conventional processors.

All of this structure becomes much clearer when viewing Figure 1, which shows the logical structure of a MOG PE's hardware environment. Each of the logical function blocks is color-coded by approximate access speed from blue (fast) to red (slow). In addition, each is marked with the associated implementation constructs in CUDA and OpenCL. The following paragraphs briefly explain the function of each block.

Accumulator. Both CUDA and OpenCL name and use GPU PE registers as the top level of the memory hierarchy. The only programmer-visible register in this function unit is the accumulator, but a number of interpreter-internal registers also are placed here.

REGs. What CUDA calls shared memory and OpenCL calls local memory is ideally the patch of memory within a SIMD engine. Although it can be accessed by any virtual PEs within that SIMD engine, GPU hardware imposes a performance penalty for violating hardware banking constraints. In MOG, this memory is partitioned along bank boundaries and used to hold the programmer-visible PE registers.

CPOOL and TEXT. The constant pool (CPOOL) and program text (TEXT) together make-up the program code. The reason they are two separate structures is a word-size difference; the CPOOL constants are 32-bit words, while instructions are just 16-bits long to maximize bandwidth utilization. Note that this is a Harvard architecture, with separate memories for code and data, so addresses in the TEXT are given in units of instructions, not bytes. In CUDA, both can be naturally implemented as 1D textures. Unfortunately, OpenCL images (the equivalent to textures) only support 2D or 3D addressing. Thus, the OpenCL version currently marks both as being stored in constant memory.

DATA. The DATA for all PEs is kept in global memory. The global memory is roughly 100X higher latency than the registers, and has banking constraints similar to those for shared/local memory. Thus, a layout respecting the banking is used to keep all PE references within the correct bank. The preferred data memory layout treats memory as a three-dimension array of 32-bit `datum_t` values: `mem[NPROC / WARPSIZE][MEMSIZE][WARPSIZE]` in which `NPROC` is the number of logical processing elements (assumed to be a multiple of `WARPSIZE`, which in turn is assumed to be a power-of-two multiple of the number of memory banks). One might have expected a two-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-two 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. One final complication is that although the DATA memory is banked for 4-byte word access, addresses are given as byte addresses and both byte and half-word operations are supported.

SYSBUF. Passing data to and from system calls is done using a separately-allocated SYSBUF space. Interpreter start-up and shut-down code handles copying between strided system buffers in the DATA space and the unit-stride SYSBUF; copying is not done by the host nor the PEs per se. Thus, the PEs operate on data using the stride that is most efficient for both CUDA and OpenCL and the host uses its native unit stride. Further, on appropriate hardware, both CUDA and OpenCL have the ability allocate memory for SYSBUF that may be slower to access, but can be directly accessed by both the PEs and host. Even if the host and PEs cannot share access without using explicit copy operations, using a separate space in global memory means that only the SYSBUF structure

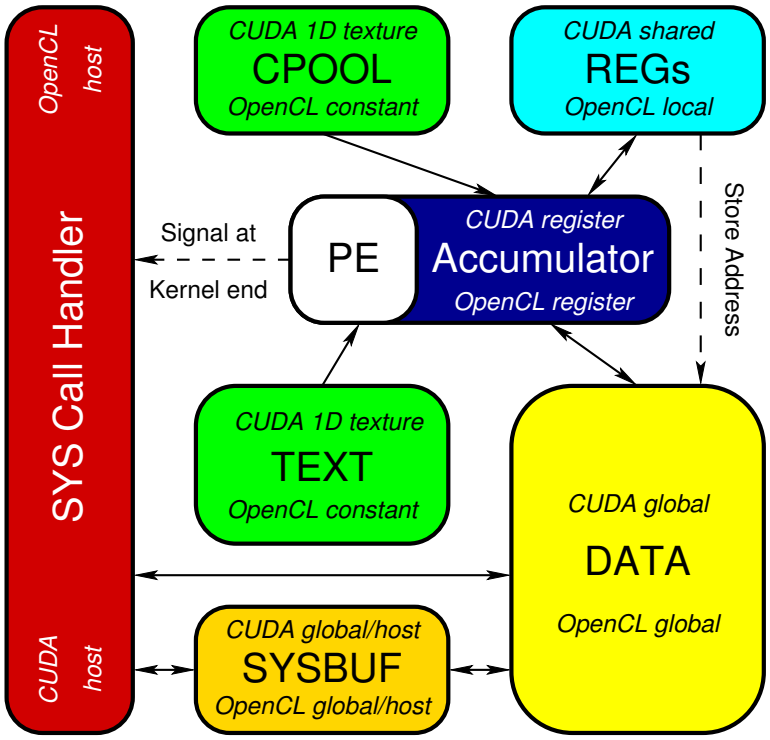

Figure 1. PE Hardware Environment and CUDA/OpenCL Implementations

needs to be copied – not various discontinuous portions of the DATA space.

SYS Call Handler. The host processor is not a processing element in our architectural model, but essentially a dedicated system call handler: a very intelligent DMA controller for interactions with various I/O devices that the PEs cannot directly access. These devices range from host main memory and disk drives to system functions and resources on other nodes in a cluster.

A PE requesting a host action will experience some delay in having its request processed. However, the significance of that delay is very dependent on the type of request being made. For example, waiting a millisecond to initiate a disk I/O request that will take 10ms to complete causes negligible loss of performance. In contrast, waiting a millisecond to send a message from one PE to another might be unacceptable.

There also is a tradeoff between delay in processing requests and improved efficiency due to aggregation of requests – the host does not need to process requests individually if requests from many PEs can be combined.

Although all of the resources available to the host processor can be made available to PEs through this interface, at this writing, only a few Posix-based system calls are implemented. Implementation of the standard MPI (Message Passing Interface) library for accessing PEs in GPUs within other nodes in a cluster has been a research interest of our group for over two years. It also is easily possible to access native GPU libraries or arbitrary user-supplied code via this system call interface with minimal overhead.

B. 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. Different GPUs have significantly different instruction sets and vendors generally discourage direct use of those instruction sets. Further, the need to have the exact same register use and control bits would make it very difficult to obtain significant factoring of instructions across MOG PEs.

The original MOG instruction set is based on a split-stack model which factors very well. The active top of the stack is kept in low-latency memory and explicit operations are used to shift portions of the stack to or from global memory as needed. This model executes efficiently and is easy to target with a custom code generator, but most existing compiler back-ends cannot easily support it.

The new MOG instruction set is designed to be much more compatible with existing compiler toolchains. It explicitly manages CUDA shared or OpenCL local memory as a set of general-purpose registers. Using a strided layout to ensure that all "registers" of each MOG PE are accessible without bank conflicts results in essentially the same access time as physical GPU registers, while indirect addressing allows factoring despite PEs accessing different registers. However, our general registers cannot be used as targets for ALU operations – a single accumulator (kept in a real GPU register) is used instead.

Why not use a more common two- or three-operand instruction format? For our interpreter, processing each operand field implies significant overhead. Compared to a three-operand format, two-operands reduce the number of field decodes

Table I
THREE, TWO, AND ONE-OPERAND FORMATS

Three Operand Format (9 operands)	Two Operand Format (8 operands)	Accumulator Format (7 operands)
		`lr $1`
`add $1,$1,$2`	`add $1,$2`	`add $2`
`xor $1,$1,$3`	`xor $1,$3`	`xor $3`
		`sr $1`
	`mov $4,$5`	`lr $5`
`and $4,$5,$6`	`and $4,$6`	`and $6`
		`sr $4`

when a source register is the same as the destination register. The single-accumulator model avoids processing yet another operand field when the result of one instruction becomes an operand to the next. These optimization opportunities occur naturally and can be made more common by register allocation and code scheduling.

The reduction in operand processing is clearly seen in the functionally-equivalent instruction sequences shown in Table I. The additional instructions for the accumulator model *always* are load and store register operations, further increasing the probability of factoring across different MIMD PEs by reducing opcode entropy.

C. Instruction Set Overview

The new MOG instruction set is summarized in Table II. Braces are used to indicate variants; for example, in the table `add{,f}` represents `add` and `addf`. Note that the "meaning" entries in the table are in most cases the actual kernel code implementations in CUDA and OpenCL.

There is a 32-bit load immediate instruction, `li`, that can be used for both integer and floating-point values – the accumulator and registers are essentially typeless 32-bit containers. To keep instruction size small, the value loaded is taken from the constant pool. The interesting twist is that the address in the CPOOL is derived by hashing the instruction encoding itself and address at which the `li` instruction is placed. The instruction field that normally would have held a register number is used to modify the hash value to avoid collisions in the CPOOL. This instruction field is set appropriately by the assembler, which also constructs the constant pool.

Most of the non-ALU instructions are loads and stores. The load register and store register, `lr` and `sr`, instructions are really moves between the accumulator and a register – and these are generally the most common instructions. The memory load and store instructions come in 32-bit word (`w`), 16-bit half-word (`h`), and 8-bit byte (`b`) variants. Half-word and byte values are sign-extended to 32 bits when loaded. Memory appears to be byte addressed, but aligned access is forced for half-word or word data by ignoring the value of the low bit or lowest two bits, respectively. (It is worthwhile noting that the current implementation does not require the OpenCL byte addressing extension.) Loads are always from memory into the accumulator, and stores always copy the value from the accumulator into the memory location specified in a register.

The arithmetic and logical operation instructions have the obvious meanings and, as in MIPS, comparison for less than is treated as an arithmetic operation. For example, `add $r` means perform a 32-bit integer addition of the integer value in register `$r` to the accumulator. Although most operations are equivalent for signed and unsigned integer operands, instructions that treat unsigned values differently have a `u` suffixed name. Likewise, 32-bit floating-point instructions end in an `f` suffix.

Although support for 64-bit floating-point operations easily could be added (and might be), the choice to omit it was deliberate. Many GPUs do not support 64-bit arithmetic and those that do provide significantly poorer performance when operating on 64-bit data. Part of the performance problem comes from the fact that a 64-bit value stored in contiguous memory locations spans two 32-bit banks. Of course, the MOG system can instead treat a 64-bit quantity as a pair of properly strided 32-bit values. However, earlier work on *speculative precision* and *native-pair* arithmetic [11] has shown that explicit treatment of two 32-bit floating-point values as a 32-bit result and a 32-bit error term can yield accuracy similar to that obtained operating on 64-bit floating-point values. Implementing many native-pair arithmetic operations requires an order of magnitude more 32-bit operations, but these operations do not require additional memory references, hence they increase the "arithmetic intensity" of a program – so the extra instructions have less impact on performance than one would expect. More important is the fact that native-pair arithmetic will work on GPUs that have no hardware support for 64-bit floating-point arithmetic. Thus, our preference is to use explicit native-pair arithmetic instead of 64-bit instructions, and ongoing research will determine if this is the correct answer.

There are three instructions that convert the value in the accumulator from one type to another. These instructions are named using the type suffixes. For example, `i2f` converts the signed integer value in the accumulator into the 32-bit floating-point number representation with the same value.

Control flow is very simple, implemented by just three instructions. The `jf` and `jt` instructions are jumps testing the value in the accumulator and conditionally jumping to a 32-bit immediate instruction address. As in the C programming language and MIPS instruction set, the value 0 is false and any non-zero value is considered to be true. The immediate value is placed in the constant pool just as it is for the `li` instruction. All other control flow is implemented by the `j` instruction, which copies the accumulator's value into the program counter.

The only remaining instruction is `sys`. In conventional architectures, there typically is a "privileged" execution mode and a group of special instructions intended for the operating

Table II
TARGET INSTRUCTION SET

Instruction	Format	Meaning
`li` *const*	*op x*	`a.i = CPOOL(pc, ir)`
`lr` $r	*op r*	`a.i = REGI(ir)`
`sr` $r	*op r*	`REGI(ir) = a.i`
`l{b,h,w}`	*op*	`a.i = MEM{B,H,I}(a.u)`
`s{b,h,w}` $r	*op r*	`MEM{B,H,W}(REGI(ir)) = a.i`
`add{,f}` $r	*op r*	`a.{i,f} += REG{I,F}(ir)`
`{and,or,xor}` $r	*op r*	`a.u {&=,|=,^=} REGU(ir)`
`div{,f,u}` $r	*op r*	`a.{i,f,u} /= REG{I,F,U}(ir)`
`mul{,f}` $r	*op r*	`a.{i,f} *= REG{I,F}(ir)`
`neg{,f}`	*op*	`a.{i,f} = -a.{i,f}`
`rem{,u}` $r	*op r*	`a.{i,u} %= REG{I,U}(ir)`
`sll` $r	*op r*	`a.u <<= REGU(ir)`
`slt{,f,u}` $r	*op r*	`a.{i,f,u} = (a.{i,f,u} < REG{I,F,U}(ir))`
`sr{a,l}` $r	*op r*	`a.{i,u} >>= REG{I,U}(ir)`
`{f,i,u}2{i,f,f}` $r	*op r*	`a.{i,f,f} = a.{f,i,u}`
`j{f,t}` *lab*	*op x*	`if (a.u {==,!=} 0) pc = CPOOL(pc, ir)`
`j`	*op*	`pc = a.u`
`sys` *func*	*op func*	`system(`*func*`)`

system. With so many relatively poorly resourced PEs, it does not make sense to suffer the overhead of having a copy of the operating system on each. Further, special operating system instructions would rarely factor, and the GPU hardware generally does not even have a hardware mechanism by which it could access most system I/O devices. Thus, `sys` simply suspends the PE until something else – typically the host – has performed the requested operation on its behalf. The function code embedded in the instruction is not intended to directly identify the desired function, but where and how the function should be processed. For example, it can distinguish between system calls needing immediate processing (e.g., an error exit) and those that can be delayed somewhat to allow more efficient aggregated processing. Similarly, it can distinguish invocations of user-supplied code or native libraries.

III. TRANSLATION OF MIPSEL ASSEMBLY CODE

By design, it should be reasonably straightforward to retarget an existing compiler toolchain to generate code for the target instruction set architecture described above. However, our goal is even more aggressive. By transforming a commonly targeted assembly language into the desired assembly language, we can effectively retarget a multitude of compiler toolchains in a single development effort. In particular, it was our goal to be able to retarget code from at least one of the major production-quality toolchains, such as LLVM or GCC.

The MIPS (Microprocessor without Interlocked Pipeline Stages) architecture began as an experimental RISC architecture at Stanford University in the early 1980s. Since then, the basic 32-bit MIPS instruction set has been extended in various ways. A multitude of commercial systems have employed MIPS-based processors – especially devices that are not computers per se, but require significant embedded processing power. The basic MIPS processor architecture also is widely used as a teaching and research tool, partly because the pipeline structure is very simple and partly because the MIPS pipeline plays a central role in a very popular architecture textbook [12].

The MIPS instruction set is unusual in a variety of ways that make it easier to transform than most other instruction sets. For example, many processors use a multitude of special condition code registers whose values are changed as side-effects of various instructions; this makes it difficult to determine which condition code settings are significant. In contrast, the basic MIPS handling of conditional expressions follows the C language convention, using explicit instructions to evaluate conditionals and set an integer register to 0 or 1. MIPS does have some unpleasant aspects, such as coprocessor-oriented floating-point arithmetic instructions, and it also is necessary to be specific about which MIPS variant is being used.

The preferred MIPS variant for our purpose is the MIPS 1 instruction set architecture with "little endian" byte order within a word. MIPS 1 is a relatively simple 32-bit model, which facilitates mapping operations to GPUs that are tuned for performance of operations on 32-bit data objects. Although most MIPS hardware can be used with either "big endian" or "little endian" byte order, the default for MIPS is generally "big endian" – which is the opposite of the byte order used by GPUs hosted by either IA32 or AMD64/Intel64 processors. Thus, the easiest MIPS variant to transform for our purpose is "little endian" MIPS 1, which is commonly known as the MIPSEL compiler target.

The GCC MIPSEL Target. Many compilers for C, C++, Fortran, and various other languages can generate MIPSEL code. While any compiler targeting MIPSEL should be usable with the system discussed in this paper, our experiments have centered on reprocessing the assembly code generated by one particular compiler.

Initially, the compiler used was LLVM's MIPSEL target. That compiler generated standard MIPS 1 instructions, but proved unable to generate valid code for floating-point operations. The problem was traced to a major flaw in the compiler's model for floating-point, which essentially resulted in the compiler confusing 32-bit and 64-bit register allocations. Rather than spending time fixing this bug, we switched to using the GCC MIPSEL target.

Beyond its wide availability and robustness, GCC is not just a compiler for a single language, but the "GNU Compiler Collection" with front-ends supporting C, C++, Fortran, Pascal, Java, Ada, etc. However, GCC's MIPSEL target code generation also proved somewhat problematic. It was relatively late in the project that it was discovered that GCC does not generate standard MIPSEL assembly language code, but uses a wide range of additional features that apparently were built-into the MIPSEL assembler used by GCC. For example, MIPS assembly language uses the `$` prefix to indicate a register name, but GCC's MIPSEL target also creates local variable names with the $ prefix. In fact, GCC also hallucinates a wide variety of instructions that are not part of MIPS 1, but that the assembler will expand. MIPS assemblers always have had various built-in macros for handy instruction sequences, so it is easy to understand how adding more might have been viewed as a harmless feature.

This proliferation of extensions to the assembly language is not harmless to our purpose. As a result, it was decided that taking an incremental approach to processing MIPSEL code would make more sense than attempting to build a highly tuned translation tool. Thus, the conversion is currently done by an easily extendable sequence of scripts, mostly using `awk`. These scripts process not just MIPSEL code, but all the extensions that we have seen GCC use in its output.

Once we are reasonably certain that GCC's MIPSEL output holds no more surprises, we intend to re-implement the processing as a proper optimizing translator. Currently, very little optimization is done during the conversion process.

The `mogcc` Script. Rather than manually running GCC to create MIPSEL code and then running a command to convert that code, we created a script that directly runs a particular version of the GCC MIPSEL compiler. This is not merely a convenience, but a hedge against different GCC versions using additional extensions to the MIPSEL assembly language that would not be understood by our current conversion process. The script performing C compile and conversion is called `mogcc`.

The latest script begins by running the GCC cross-compiler version 4.1.1 for the "mipsel-linux" target. Important command line options given to GCC include those listed in Table III.

The first three are critical in forcing the expected MIPSEL instruction set variant. Note that neither the DSP nor 3D extensions can be enabled. The GPU PEs have very limited low-latency memory, so the decision was made to avoid reserving a word for a frame pointer.

After invoking GCC, `mogcc` uses `sed` and `awk` scripts to simplify and normalize the MIPSEL code. Various unnecessary directives and comments are removed, symbolic register names are converted into their numerical equivalents (e.g., `$ra` becomes `$31`), the `$` prefix of compiler-generated labels is converted into `DOL_`, etc. using `sed`. The `awk` scripts perform higher-level cleaning and normalization of the code. For example, the cleaning involves tasks such as removal of global offset table references, conversion of MIPS multiply and divide sequences into more conventional instructions, and elimination of offsets in load and store instruction address references. The normalization largely concerns conversion of floating-point coprocessor operations into instructions of the same form as the integer instructions.

The `mogcc` script then passes the cleaned and normalized code through several conversion passes, as described in the following sections.

The `conflow` Script. Control flow in MIPS is relatively straightforward, consisting of just conditional branches, unconditional jump, jump register, and two forms of jump-and-link. However, MIPS assemblers traditionally offer additional operations implemented as sequences of actual MIPS instructions, and the code output by GCC extends that apparent instruction set even further. The purpose of the `conflow` script is to convert all the control flow into simple jump-true (`jt`), jump-false (`jf`), and jump-register (`j`) instructions.

Converting all the conditional branches into true or false jumps is straightforward. A large fraction of the conditional branches are testing if the value in a register is equal to the value in MIPS register `$0`, which is hardwired as zero – the only false value in the C language. All of these branches can be directly coded using `jt` or `jf` without wasting a register to hold zero. Branches testing equality of two non-`$0` registers can be converted into an `xor` of those registers followed by `jt` or `jf`, as appropriate.

Unconditional branches (a GCC extension) and jumps are simply translated into jump-register instructions.

The jump-and-link instructions are functionally unconditional jumps preceded by placing the return address into the return address register, `$31`. The return address is set by inserting a label and loading its value.

The `twe2acc` Script. Typical of RISC architecture instruction sets, MIPS arithmetic operations generally use a three-register form. Each such instruction encodes two source registers and a destination register. Alternatively, the second source register can be replaced with a short 16-bit immediate value. However, this encoding is not efficient for MOG.

The `twe2acc` script performs the straightforward conversion of three-operand MIPS instructions into single-accumulator code. It is significant to note that the single accumulator is accompanied by a set of general-purpose registers. Thus, as shown in the example of Table I, the explicit operand for each instruction is generally a register – not a memory address, as older single-accumulator instruction sets would have required.

The `lrsropt` Script. Although conversion from MIPS three-operand into a single-accumulator with general registers model offers the potential to improve performance, the code generated by `twe2acc` contains many unnecessary `lr` and `sr` instructions. The `lrsropt` script attempts to remove redundant `lr` and `sr` instructions. The analysis used is very simple. There is no need to load a value just saved, nor is there a need to save a value just loaded from the same register. This simple analysis removes most of the unnecessary register accesses.

The `regmap` Script. The greater the fraction of low-latency GPU memory allocated to each virtual PE, the fewer virtual PEs can be co-resident. Typically, global memory latency dominates execution time such that increasing the number of

`-mips1`	Select the MIPS 1 architecture model
`-mfp32`	Force floating-point registers to be 32-bits wide
`-msym32`	Force all symbols to be 32 bits wide
`--omit-frame-pointer`	Disable use of a frame pointer register

Table III
RELEVANT MIPSEL GCC OPTIONS

co-resident virtual PEs roughly multiplies the work done per unit time, so transformations that result in use of less low-latency memory per virtual PE produce a proportionate parallel speedup.

The basic MIPS instruction set has 32 user-visible integer registers and various other special and coprocessor registers. Blindly allocating enough low-latency memory for each PE to have all of these registers would result in a tiny virtualization factor, very limited parallelism, and correspondingly poor performance. The purpose of the `regmap` script is to minimize and prioritize the set of registers needed; the following MIPS registers are treated specially:

- `$0` In MIPS, this is `$zero` – a hardwired 0 value. It is not used in MOG.
- `$28` In MIPS, this is `$gp` – a pointer to a 64K-byte global constant pool that also can be used to store small global variables. The `mogcc` script removes all references to `$28`, replacing constant references with immediate loads and allowing variables to be allocated normally.
- `$29` This is `$sp` – the stack pointer. To preserve the calling conventions of MIPSEL, the stack pointer is always allocated as MOG register `$0`.
- `$30` In MIPS, this is `$fp` – the frame pointer. Since a frame pointer is not strictly necessary, `mogcc` requests GCC generate code that does not use it and it is not preallocated. However, if used in the MIPS code, it will be allocated.
- `$31` This is `$ra` – the return address register. To preserve the calling conventions of MIPSEL, the return address is always kept in MOG register `$1`.

Other registers, including the coprocessor floating-point registers, are allocated space by `regmap` only if they are used.

However, it is not necessarily true that all used registers must be allocated registers in the MOG system. Thus, `regmap` not only detects which registers are used, but counts their usage frequency. After pre-allocating registers $0 and $1, registers are renumbered in order of decreasing frequency of use. A future version of the system is expected to intelligently convert higher-renumbered registers into variables in global memory as well as performing more sophisticated analysis and transformations to minimize MAXLIVE.

The `colseg` Script. As a result of converting code constructs such as `$gp` references and `.comm` references into definitions of data, the transformed code now freely mixes both code and data. The purpose of the `colseg` script (collect into segments) is simply to separate the data and code into two segments. First, all data is defined in a single data segment. Then, all code is defined in a single code segment.

The first data defined are not derived from the MIPSEL assembly language, but are built-in portions of the MOG environment. The first word is `NPROC`, the total number of PEs in the system. The second word is `IPROC`, the number of this PE, as an unique integer value between 0 and `NPROC`-1 (inclusive). The next few data objects are defining the buffers for interactions with system calls.

The `deadopt` Script. The `deadopt` script implements dead code and dead store removal using the standard analysis. The processing takes advantage of the fact that data segments and labels in data space no longer are intermingled with code, which is why `colseg` must be executed before it.

IV. THE MOG ASSEMBLER & INTERPRETERS

The assembler is a fairly straightforward C program implementing conventional multi-pass assembly. The output is not an object file, but rather a set of header files that define various parameters, the program data and text segments, and even the interpreter structure. The CUDA and OpenCL versions are really just wrappers into which the assembler-generated customized structure is inserted. These wrappers repeatedly invoke the `emulate()` kernel, checking the SYSBUF contents after each completion to determine what system calls have been requested. Previous research determined that the overall best MOG interpreter structure is a sequence of single-instruction subinterpreters created specifically for the application program [8], and the new system is a re-implementation of this approach.

The interpreter loop sequences through subinterpreters, each of which compares the current instruction to the opcode it can process, and interprets the instruction if the opcode matches. A very simple version of this approach was used for Genetic Programming on a GPU [13], and Nilsson and Tanaka's original concept [14] does somewhat better by picking 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 `LR`, `ADD`, `XOR` then the instruction sequence `LR`, `ADD`, `XOR` would take only one cycle – but `ADD`, `LR`, `XOR` would take 2 cycles and `XOR`, `ADD`, `LR` would take 3. In our earlier work targeting the MasPar MP1 [15], 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. [16] 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.

The assembler uses frequency estimates of both individual instructions and *instruction digrams* to create an optimized sequence of single-instruction interpreters. By default, the

assembler uses static analysis to obtain these estimates, but they also can be obtained by tracing one or more program executions on test data. The assembler first creates a sequence of single instruction subinterpreters in which each instruction type used in the program occurs at least once, and frequency of occurrence of subinterpreters for a particular instruction type mirrors the frequency with which that instruction is expected to execute. The assembler then uses a modified evolutionary search 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 36 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 3 and 6.

Space does not permit detailed discussion of the interpreters or their performance, but interpreter performance using CUDA on the same GPUs is not worse than the split-stack system achieved [8]. On the same NVIDIA hardware, the OpenCL version appears to be slightly slower than the CUDA version. The OpenCL version compiles on AMD/ATI hardware, but hangs for some PE virtualizations; where the system works, performance is again comparable. Overall, one can expect less than an order of magnitude slower execution than tuned native code on the same GPU. Wildly dynamic MIMD code, including recursion, also works with apparently good efficiency.

V. Conclusions

Earlier work showed how it is possible to efficiently execute MIMD On GPU (MOG) via interpretation, but failed to provide a practical method by which existing compilers could be used. The new accumulator/register instruction set described here, combined with translation from MIPSEL assembly code, solves this problem. The new system also can generate portable OpenCL interpreter code, not just CUDA, and has run on both NVIDIA and AMD/ATI hardware (although not without issues).

A full source code version of this system targeting only CUDA has been freely available from `Aggregate.Org` since November 2010. The complete version including OpenCL support will be posted later this year. Future work includes replacing the MIPSEL conversion scripts with an optimizing translator and implementation of various system calls, especially those supporting MPI message passing between PEs within and across GPUs.

References

[1] ClearSpeed, "ClearSpeed whitepaper: CSX processor architecture," *ClearSpeed Technology plc*, vol. PN-1110-0702, 2007.

[2] NVIDIA, "NVIDIA CUDA compute unified device architecture programming guide version 1.0," June 2007.

[3] ATI, "ATI stream SDK user guide v1.3-beta," December 2008.

[4] Message Passing Interface Forum, "MPI: A Message Passing Interface," in *Proceedings of Supercomputing '93*, pp. 878–883, IEEE Computer Society Press, 1993.

[5] T. Blank, "The MasPar MP-1 Architecture," *35th IEEE Computer Society International Conference (COMPCON)*, February 1990.

[6] 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.

[7] 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.

[8] H. Dietz and B. Young, "MIMD Interpretation on a GPU," in *Languages and Compilers for Parallel Computing* (G. Gao, L. Pollock, J. Cavazos, and X. Li, eds.), vol. 5898 of *Lecture Notes in Computer Science*, pp. 65–79, Springer Berlin / Heidelberg, 2010.

[9] C. Lattner, "LLVM: An Infrastructure for Multi-Stage Optimization," Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[10] Khronos OpenCL Working Group, "The OpenCL specification version 1.0," December 2008.

[11] H. Dietz, B. Dieter, R. Fisher, and K. Chang, "Floating-point computation with just enough accuracy," *Lecture Notes in Computer Science*, vol. 3991, pp. 226 – 233, Apr 2006.

[12] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 4th ed., 2008.

[13] 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.

[14] 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.

[15] 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.

[16] 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.