Enabling OpenCL on a Configurable, VLIW Chip-Multiprocessor

Samuel J. Parker, Vassilios A. Chouliaras

Abstract—The slow-down in Moore’s law and an ever increasing computation requirements in the scientific, as well as consumer, domains has required a shift in computer system architectures and subsequent programming paradigms. In the last decade we have moved from single-core CPUs, to multi-core system-on-chips (SoCs), with the use many-core accelerators becoming more commonplace. This new paradigm has required the design, and introduction, of new languages such as CUDA, OpenCL and OpenACC. This paper presents an initial OpenCL driver for a custom, configurable, VLIW chip multiprocessor (CMP).

Index Terms—OpenCL, VLIW, hardware/software co-design, reconfigurable hardware.

I. INTRODUCTION

For many years, PCs have contained powerful graphics cards (GPUs) used for 3D design and gaming. These processors are designed to compute millions of pixels, several times a second using highly data- and thread-parallel algorithms. Now in the post-pc era, the growth in the high pixel density displays has driven rapid growth in mobile GPU development as well. The key difference between CPUs and GPUs is that CPUs are primarily designed for single-threaded performance of relatively complex programs that interact with the user and utilise system libraries. To do this, CPUs have a few, highly-complex, cores, each running one or two threads, generally allowing a user to run an operating system and multi-task between programs. GPUs however are designed to perform the same calculation across millions of data elements as a throughput device, as it is not important that one pixel needs to be calculated before another. GPUs perform such an operation by utilising 10’s of simple SIMD cores, each capable of running hundreds of threads.

To harness the power of their GPUs for computations other than graphics, NVIDIA modified their GPU architecture and released a proprietary language called CUDA to support it. In 2012, the most powerful supercomputer used a mix x86 CPUs and NVIDIA Tesla accelerators. To enable the use of accelerators, in a non-vendor specific way, the Kronos Group released an open standard called OpenCL [1].

OpenCL programs are comprised of two parts: the host, and the device code. The host code is written in standard C or C++, whereas the device code is written in a modified version of C which has been extended with some attributes, data types and built-in functions. Device code is written as a function called a kernel; this is an expression of a thread-parallel unit of work. Kernels can be compiled offline, or more effectively, at runtime when the target device is known - this allows for portability and greater optimisations. The host program uses an installed OpenCL library to query the system for any compatible OpenCL devices for which it can create a program for. The user has to explicitly manage data between the host and the device; this requires creating buffers on the device and writing host data to them.

The purpose of our chip multiprocessor driver (CMPCL) is to be compliant OpenCL (version 1.1) driver which enables optimally running kernels on CMP accelerators, specifically the LE1.

II. LE1

The LE1 [2] is a parametric VLIW CMP, its instruction set architecture modeled on an integer DSP. It is capable of issuing multiple operations per clock cycle; these are bundled into long instruction words. The processor is configurable by the number of cores in the CMP configuration, with the 8-stage pipelined core of the processor, depicted in Fig 1, is parameterised by:

- how many RISC instructions are dispatched each cycle,
- the number of arithmetic logic units (ALUs) per processor,
- the number of integer multipliers per processor,
- latencies and memory configurations.

Fig 1. Block diagram showing the pipeline of the LE1.
III. CMPCL

The driver is comprised of several parts:

• The front-end, which implements all the functions of the OpenCL API; containing event queues, kernel objects, program objects, memory objects and device targets.

• The compiler which transforms, links and optimises the device source code into machine code.

• The back-end holds device specific data and issues all the final commands to the device, though the device is currently only simulated, this includes managing the data transfers and coherency and setting up the simulator.

A. Front-end

The front-end is based off the Clover project (which is now the OpenCL state tracker in Gallium3D) implementing the OpenCL API supporting an embedded profile; meaning we do not currently support 3D image or 64-bit data. The front-end holds the target objects to which the host program can query the platform and the variety of devices available, assign a device to a context, create a command queue and start pushing commands to it.

B. Compiler

We use Clang libraries for our compiler, as do many graphic drivers; it is a front-end compiler to the LLVM framework [3] and is designed as a set of libraries allowing developers to create standalone programs from them. When invoked by the frontend, the compiler checks that the device code is valid and prepares it for the later stages; including macro expansion and function inlining. When invoked by the back-end, the libraries are used to perform source-to-source transformations using the abstract syntax tree (AST) and also link the final kernel with a small function that launches separate workgroups. The device code is finally compiled into machine code using the compiler back-end we have created for LLVM.

C. Back-end

The back-end consists of a device object which has an associated thread that pulls events off the internal queue and responds to them. Here, front-end objects, such as kernels, are used to create device specific ones. The back-end maintains coherency between the LE1’s simulated memory and the buffer objects as well as controlling the simulator. When the back-end receives a clEnqueueNDRangeKernel event, it invokes the compiler to create the final code.

IV. INITIAL RESULTS

We are using integer benchmarks, primarily from the Rodinia suite[4][5]; these are generally without SIMD data and contain control flow as well as frequent use of thread synchronisation points. The breadth-first search performs the operation over a graph containing 4096 nodes; the results are shown from Fig 2 through to Fig 5. The sorting benchmark is taken from SHOC [6] and sorts 32KB of data, with a radix of 4, in ascending order – the results shown in Fig 6 to Fig 9. Both of these algorithms use multiple kernels, and run each kernel several times. The results show the average number of cycles taken for each kernel to complete on the LE1 and does not consider the total program time. The microarchitecture is kept at a dual-issue width during the benchmarks where we vary core count.

Fig 2. Average execution cycle count of the first kernel for BFS using a varying number of cores.

Fig 3. Average execution cycle count of the second kernel for BFS using a varying number of cores.

Fig 4. Average execution cycle count of the first kernel for BFS using multiple issue widths in a single core configuration.

Fig 5. Average execution cycle count of the second kernel for BFS using multiple issue widths in a single core configuration.
V. Conclusion

Although our current number of completed benchmarks is small, they do show that it is possible to run OpenCL kernels on the LE1, through the use of source-to-source transformations. The results also show that the execution times decrease linearly with the increased number of cores, confirming the advantages of an explicitly thread-based language. Though, with the current state of the compiler, we are not able to effectively utilize fine-grained instruction level parallelism of wider issue machine configurations. This should improve as we enlarge the instruction scheduling regions, from basic blocks to extended basic blocks, and improve the microarchitecture details available to the backend. The driver also needs to be extended to exploit the configurable nature of the LE1, by using static analysis and profiling to make decisions on the most effective microarchitecture configuration.

REFERENCES