Investigation of Parallel Programmability and Performance of a Calxeda ARM Server Using OpenCL

David Richie[1], James Ross[2], Jordan Ruloff[2], Song Park[3], Lori Pollock[4], and Dale Shires[3]

[1] Brown Deer Technology, Forest Hill, MD
drichie@browndeertechnology.com
[2] Dynamics Resources Corporation (DRC), Andover, MA
{jaross,jruloff}@drc.com
[3] U.S. Army Research Laboratory, Aberdeen, MD
{song.j.park.civ,dale.r.shires.civ}@mail.mil
[4] University of Delaware, Newark, DE
pollock@udel.edu

Abstract. This paper explores the parallel programmability, performance, and energy efficiency of a recently available Calxeda ARM-based server as a potential energy-efficient platform for computationally intensive applications. A novel OpenCL-based parallel programming model for the Calxeda ARM server is achieved via the use of a higher level STandarD Compute Layer (STDCL) application programming interface and a remote procedure call (RPC) implementation. Empirical measurements of the performance of the platform are obtained and presented using an N-body code executed in various configurations. Furthermore, an auto-tuning technique was developed and analyzed for optimization of the N-body algorithm on a specific architecture.

Keywords: High Performance Computing, ARM server, OpenCL.

1 Introduction

Power consumption has arguably become the single most critical factor impacting high performance computing (HPC) for architectures ranging from large-scale supercomputers to data centers to mobile computing platforms [9,3,6]. It is within this context that ARM processors, which dominate the mobile smartphone and tablet market, are being repurposed as an alternative to the x86 processors that have dominated HPC, desktops, and workstations, for well over a decade. Whereas a modern x86 processor is based on a CISC architecture with substantial per-core capability, ARM processors are based on a RISC architecture designed for low-power operation [8].

Newly developed ARM-based servers, such as those from Calxeda, offer low-energy usage, high-memory density, high-core density, and high-storage density–all qualities that motivate using these servers for HPC applications. However, the relative core efficiency as compared with x86 CPUs in terms of cost and power metrics remains an open question. The Calxeda ARM servers [5] represent the most direct attempt to date aimed at bringing ARM processors into competition with traditional x86 servers.

This paper reports on exploration into the parallel programmability, performance, and energy efficiency of a Calxeda ARM server using an OpenCL-based approach. OpenCL provides an explicit low-level programming model for co-processor architectures that may also be used to exploit the parallelism of systems with multi-core CPUs treated as abstract OpenCL compute devices. The OpenCL host node is treated normally, while the rest of the nodes are virtualized and treated as OpenCL compute devices through a Remote Procedure Call (RPC) OpenCL implementation. This specialized RPC-based OpenCL implementation permits existing multi-device OpenCL codes to execute across multiple networked compute devices.

Section 2 describes the Calxeda server system used for evaluation. Section 3 investigates the use of existing software packages that leverage OpenCL to provide a novel parallel programming model for platforms like the Calxeda ARM servers. Section 4 discusses the use of an N-body benchmark, including auto-tuning parameterizations, to measure the performance of the Calxeda ARM server. Section 5 provides the evaluation of the Calxeda ARM server. Section 6 presents related work on OpenCL performance. Finally, Sect. 7 closes with conclusions and future work.

2 The Calxeda ARM Server

The Calxeda ARM server experimented for this study was provided by Exxact Corporation and accessible from a remote desktop allowing direct access to a login node connected to the Calxeda system. It was configured with two Calxeda EnergyCards, each containing four quad-core ARM Cortex-A9 system on chip (SoC) devices, as shown in Fig. 1, acting as independent nodes (device nodes); thus, a potential total core count of $2 \times 4 \times 4 = 32$ cores. However, the server that was provided had five device nodes, or 20 cores, configured and accessible for use. Based on documentation, the ARM cores operate at a clock frequency in the range of 1.1 GHz to 1.4 GHz and have access to 4GB DDR3 ECC memory per device node. It is important to note that each SoC interacts exclusively over the integrated network fabric and the quad-socket card is not a shared memory platform like a conventional server motherboard.

Each quad-core node was accessible via Secure Shell (SSH) and presented itself as a standalone node running its own operating system (OS) image, specifically, Ubuntu 12.10. The environment is a typical Ubuntu OS system and included standard development tools, e.g., GCC 4.7, along with most of the supporting tools needed for compiling code. The platform included a baseboard management controller (BMC) that allowed power measurements to be obtained for each ARM device node.

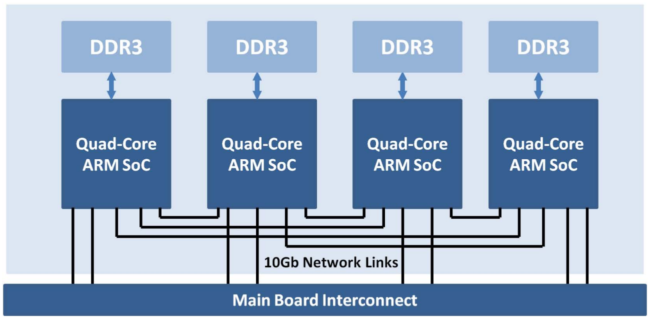

Fig. 1. Block diagram of the Calxeda EnergyCard showing the basic platform architecture of the Calxeda ARM server. Each card contains four quad-core ARM Cortex-A9 SoCs with dedicated banks of 4GB DDR3 EEC memory. Each ARM SoC has an integrated fabric switch with five (5) 10Gb network links supporting a direct all-to-all on-card topology and a total of eight (8) links exported to the main system board to connect with additional cards.

3 Parallel Programming Approach

This investigation leverages freely available open-source software packages to support an OpenCL-based parallel programming model for the Calxeda ARM server. In particular, the CO-PRocessing THReads (COPRTHR) software development kit (SDK) [1] was used to provide libraries and tools for the investigation, including an OpenCL implementation for the quad-core ARM processors, an OpenCL RPC implementation for networked compute devices, and an implementation of STandarD Compute Layer (STDCL).

OpenCL is an industry standard application programming interface (API) for parallel programming of heterogeneous computing platforms [15]. OpenCL provides a portable vendor- and device-independent low-level API for programming parallel platforms. Since an OpenCL implementation for the ARM processor was not available as part of the standard software stack on the Calxeda system, the implementation provided by the COPRTHR SDK was selected to test the ARM processor. In addition, the OpenCL RPC implementation provided by the COPRTHR SDK, designated CLRPC, provided the ability to target multiple ARM SoCs as networked compute devices on the Calxeda platform, which allowed the investigation of inter-device parallelism within an OpenCL programming model.

The operation of CLRPC is outside of the OpenCL standard and warrants discussion. The OpenCL compute capability of any networked device can be exported using a CLRPC server (clrpcd) running on the host node of the device. A clrpcd server will export all available OpenCL platforms on a given host node. An application code running elsewhere can access all exported OpenCL platforms through libocl.so by providing a list of the network addresses for known servers in an ocl.conf file that replaces the standard OpenCL installable client driver (ICD) enumeration.

One drawback of OpenCL is that, in comparison to its closest vendor-supported competitor, Compute Unified Device Architecture (CUDA), its direct use in application development proves to be more tedious and complicated. For this reason, STDCL is used here for host-side application programming. STDCL is an API [4] that leverages OpenCL, but supports more natural programming syntax and semantics for application development. STDCL provides support for default compute contexts, conventional memory allocation of device-sharable memory, event management, and a dynamic kernel loader that supports a more traditional compilation model and an offline kernel compiler.

A particularly significant feature of STDCL used in this investigation is the default context, stdnpu, that leverages CLRPC to provide a single compute context containing all networked compute devices. This feature has no direct equivalent in the OpenCL standard and creates a unique parallel programming model for the Calxeda platform.

4 Auto-Tuning OpenCL N-Body Benchmark

The N-body algorithm is used to solve Newton's laws of motion for N particles subject to an inter-particle force. The algorithm requires the update of all particle positions and velocities based upon the distance of a given particle to all others by calculating a distance-dependent force and then numerically integrating the equations of motion using a fixed-time step. The N-body algorithm provides an excellent benchmark[12] for the evaluation of a computing platform for several reasons. First, the basic algorithm is representative of many real-world computational kernels, and may serve as a proxy for their expected performance. Second, the manner in which the simulation is performed is relatively clean without superfluous computations that would complicate the interpretation of the performance benchmarks. Third, the algorithm provides a simple mechanism of sweeping a single parameter, the number of particles, to drive the system into a compute bound regime because computation and data movement scale as $O(N^2)$ and $O(N)$, respectively. Finally, the simulation is commonly implemented on a range of architectures and provides a convenient canonical algorithm for comparative benchmarking.

Although OpenCL software is portable across many devices, that does not imply performance portability. This investigation included running an auto-tuning OpenCL N-body benchmark that parameterized the kernel in ways that impact performance across different architectures and compilers. The parameters are used to auto-generate kernel source, as opposed to passing arguments, compiled for each unique case. Similar brute force automatically tuning schemes have been used to optimize linear algebra routines in the ATLAS and GATLAS packages for particular architectures.

The kernel parameterization is described below. The parameter *nmulti* is the number of particles updated per thread. This outer-loop multiplicity may allow a compiler to automatically vectorize the computation. The parameter *nunroll* defines the explicit unrolling within the inner loop over particle pair interactions.

The parameter *nblock* is the number of particle positions cooperatively cached by a work-group in local memory for calculating particle pair interactions. This essentially replaces the inner loop over particles with a double loop over blocks of *nblock* particles and a nested loop over the cached particle positions. The parameter *nthread* determines the OpenCL work-group size.

5 Evaluation Study

5.1 Methodology

The software stack supporting an OpenCL-based parallel programming model described in Sect. 3 had never been tested on the Calxeda ARM server architecture. The simplest configuration used to obtain benchmarks involved the execution of an OpenCL benchmark using the CPU OpenCL implementation directly (OpenCL direct). This scenario involves running the host program on the same device node that is also used to execute the OpenCL kernels on up to four cores, and is the conventional model for OpenCL CPU implementations.

CLRPC enables a transparent client-server model to export an OpenCL compute device to another node. Using this scenario (CLRPC remote), a clrpcd server is run on a different device node from that which is used to execute the host program. This configuration is still limited to using up to four cores for the execution of OpenCL kernels.

A variation of the CLRPC remote configuration involves executing the host program on the same device node used to run the clrpcd server, and in turn execute the OpenCL kernels (CLRPC local). The difference here between OpenCL direct and CLRPC local is that in the latter case, the device must incur the overhead of interacting with the local clrpcd server.

All of the above configurations are limited to using up to four cores for executing OpenCL kernels. The STDCL compute context stdnpu, that leverages CLRPC to provide access to all networked compute devices, expands the parallel programmability of the platform by allowing multiple compute devices to be used from a single compute context. This configuration (stdnpu) can be used to study the scaling of the parallel programming model over the entire system of networked ARM SoCs in a Calxeda server, aggregated into a single compute context accessible from a host program running on any node.

Performance measurements reflect results for an OpenCL N-body benchmark based on default code originally tuned for high-end GPUs. This code was modified to develop the auto-tuning benchmark described in Sect. 4. This benchmark was used with different configurations and parameters to explore the Calxeda ARM server platform and the OpenCL-based parallel programming model presented in Sect. 3.

The default benchmark was first used on a single device node with the conventional OpenCL configuration involving a STDCL host program using the OpenCL platform implementation directly (OpenCL direct). The auto-tuning version of the benchmark was then employed to identify the optimum parameters for the parameterized kernel, anticipating that the search would improve the measured performance of the algorithm.

Performance measurements were obtained using CLRPC, which provides a key component for expanding the parallel programmability of the platform beyond a single quad-core device. To address the issue of networking overhead, measurements were taken by running the CLRPC server and STDCL host program on the same device node (CLRPC local) and then repeated by running the CLRPC server on a different device node (CLRPC remote).

Finally, the scaling of the benchmark for 4 to 16 ARM cores was studied using stdnpu, the STDCL default context for all networked compute devices that, in turn, leveraged CLRPC.

Power is measured using the BMC interface for more than the quad-core ARM processor itself and includes other elements of the SoC like the network fabric, global memory, etc. Power data consisted of measurements at idle and under full load while running an N-body benchmark with 32,768 particles, so that the device would be driven well into the compute regime and under full utilization.

5.2 Results

The results reported in this paper are for single-precision floating point computations. The default N-body benchmark was executed with 16,384 particles and nthread=16 using the OpenCL direct configuration and exhibited 983 MFLOPS (million floating point operations per second). With this number of particles, the simulation is driven well into a compute bound regime on this architecture, as discussed more thoroughly below. Examining CPU utilization using the Linux top command showed nearly 400% utilization, indicating that all four ARM cores were utilized at full load. The auto-tuning benchmark was run using the same number of particles and identified an optimized kernel that exhibited 1,094 MFLOPS, a gain of 11.3% over the default benchmark.

Using the CLRPC local configuration, where the clrpcd server and STDCL host program are run on the same device node, the identical default benchmark exhibited 840 MFLOPS. Examining CPU utilization showed that the clrpcd server was running at approximately 350% with the host program using the balance of approximately 50%. With this configuration, the clrpcd server executes the computational load, and the host program drives the computation through RPC client calls. The results provide an indication of the host program overhead incurred when using CLRPC. The reduction in performance (15%) is consistent with the reduction in utilization (13%).

Using the CLRPC remote configuration, where the clrpcd server and STDCL host program are run on different device nodes, the identical benchmark achieved 980 MFLOPS and the clrpcd server showed nearly 400% utilization. This demonstrates that the resource contention found with the CLRPC local configuration can be entirely mitigated when the clrpcd server is run on a separate device node and the CLRPC overhead remains with the host program.

Figure 2 portrays the results for different numbers of particles. Using the OpenCL direct configuration, the benchmark reaches a pure compute regime at around 1,024 particles. For smaller numbers of particles, the performance is negatively impacted by non-compute operations. This reduced performance for small numbers of particles is observed on nearly all architectures and is expected.

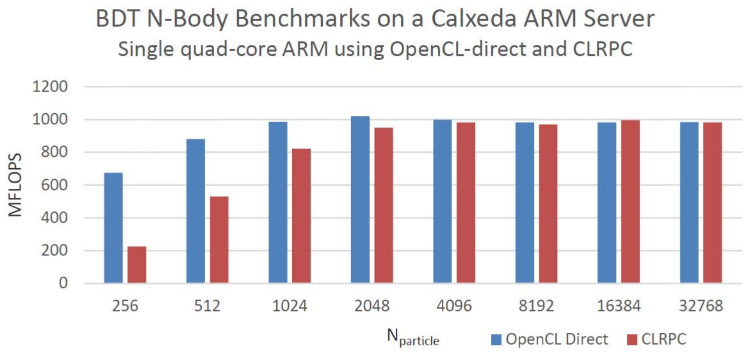

Fig. 2. Performance results for BDT N-body on a single quad-core ARM device on a Calxeda server. Comparison is between the OpenCL direct and CLRPC remote configurations.

Results using the CLRPC remote configuration are also depicted in Fig. 2, where it can be seen that a greater overhead is incurred for small numbers of particles. Not only does the benchmark show that it reaches a compute regime at a slightly larger number of particles by roughly a factor of two, there is a greater decrease in performance as the number of particles is reduced.

Two factors are likely impacting the performance. First, the relative cost of data movement and compute will decrease linearly with the number of particles, being $O(N)$ and $O(N^2)$, respectively. Second, there is is an overhead associated with the CLRPC implementation that will become more pronounced as the computational load is reduced. Nevertheless, the results show that CLRPC can be used effectively for computationally intensive tasks.

The default STDCL context stdnpu containing all networked compute devices, which leverages CLRPC, was used to demonstrate the parallel programmability of the ARM server beyond a single quad-core processor. Figure 3 represents the scaling of the OpenCL N-body benchmark using multiple ARM SoCs up to 16 ARM cores for various system sizes. For the smallest system, scaling is predictably poor because insufficient work is provided per core to mitigate the overhead. However, with a sufficiently large system, the scaling is nearly linear through 12 cores. The deviation from linear scaling at 16 cores is attributable to the overhead described above for the CLRPC local configuration, since the 4th ARM SoC is executing the host program.

Power measurements are displayed in Fig. 4 and indicate a power variance between ARM SoCs that is significant, with a minimum (maximum) idle power of 3.3 W (4.6 W). Under full load, the power increase is reasonably consistent with a minimum (maximum) cost of an additional 0.85 W (1.2 W).

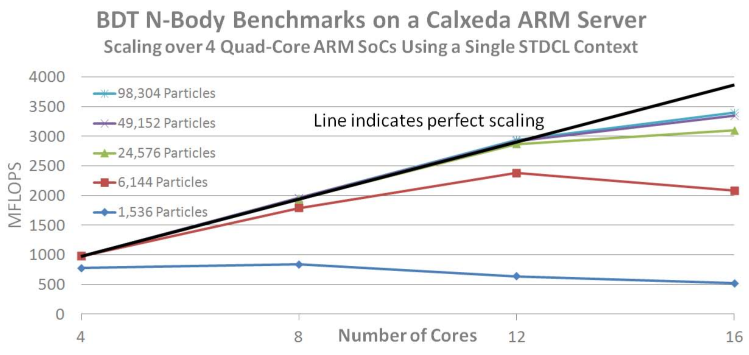

Fig. 3. BDT N-body benchmark scaling up to 16 ARM cores using a single STDCL compute context, stdnpu, containing four (4) networked ARM SoC compute devices. STDCL uses CLRPC to access all of the ARM devices in the Calxeda platform.

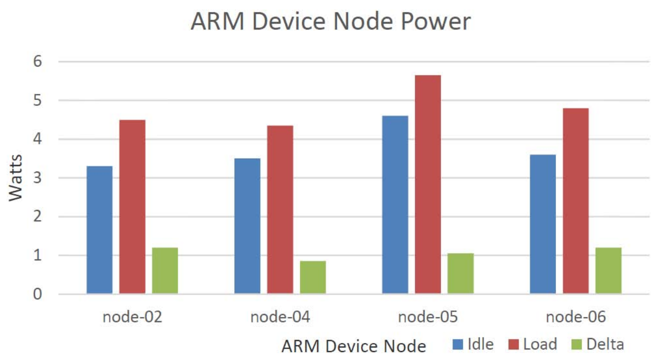

Fig. 4. Calxeda power requirements at idle (blue) and under load (orange) while executing N-body application. The difference between idle and load are indicated by the green bar.

5.3 Analysis and Discussion

The Calxeda ARM server provides a platform that is functionally equivalent to a small cluster of quad-core processors, each running a standard Linux OS. Using an OpenCL N-body benchmark, the ARM processor yielded the computational performance of approximately 1 GFLOP. The ARM SoC device nodes used 3.3 W-4.6 W of power when idle and 4.35 W-5.65 W under full load. Therefore, the power efficiency for this benchmark is approximately 1 GFLOPS/W in terms of processor power and approximately 0.2 GFLOPS/W in terms of total node power. By comparison, using the auto-tuning N-body benchmark on a system with dual Intel Xeon X5650 CPUs (12 cores @ 2.66 GHz) resulted 83.7 GFLOPS with a combined thermal design power of 190 W. Based on this limited scope of power data, it remains difficult to assess the relative power efficiency of the quad-core ARM SoC device since the measured power includes components that do not align with the power measurements of other architectures.

The Calxeda system supported the use of CLRPC to allow OpenCL applications to access any networked OpenCL devices. CLRPC performance shows some overhead for workloads with lightweight kernels, and matches the performance of using OpenCL directly for computationally intensive workloads.

6 Related Work

Others have developed test suites that cover combinations of OpenCL operations, element types, and local sizes to test performance and numerical precision of GPU systems in terms of OpenCL operations [10].

In comparative performance studies between CUDA and OpenCL versions of the same applications, Fang et.al [7] and Komatsu et.al. [11], have shown that performance can be comparable if the kernels are optimized by hand or by compiler optimizations. They also highlighted how automatic parameter tuning is essential to enable a single OpenCL code to run efficiently on various GPUs, motivating the need for auto-tuning for each system and for comparative performance studies. Yao et.al. [16] studied the performance portability of OpenCL across diverse architectures including NVIDIA GPU, Intel Ivy Bridge CPU, and AMD Fusion APU, using three OpenCL benchmarks–SGEMM, SpMV, and FFT. They found that performance portability requires tuning threads-data mapping, data layout, tiling size, data caching, and operation-specific factors. Other studies of OpenCL performance include comparing against OpenMP for multi-core CPUs [14]. Beyond CUDA and OpenCL, frameworks for GPU programming with abstraction include Halide [13] and HIPAcc [2] for image processing.

7 Conclusions and Future Work

To our knowledge, this is the first effort to investigate the use of OpenCL to support a parallel programming model for a Calxeda ARM server. The initial success of applying a novel OpenCL-based parallel programming model that includes the use of a high-level abstraction for OpenCL and an RPC implementation of OpenCL to access networked compute devices demonstrates the potential of this model for utilizing the large number of ARM cores available on a fully configured Calxeda system. Empirical results are obtained for executing an OpenCL N-body benchmark using various configurations and for a range of parameters from which the power efficiency of the quad-core ARM processors could be evaluated. An auto-tuning benchmark was developed that can be used to optimize the computational kernel for a given architecture. Power efficiency results are not sufficient to make a determination as to the energy efficiency of an ARM-based server compared with competing architectures. Future work includes scaling to a much larger number of cores.

Acknowledgments This research was supported in part by an appointment to the Faculty Research Participation Program at the U.S. Army Research Laboratory administered by the Oak Ridge Institute for Science and Engineering through an interagency agreement between the U.S. Department of Energy and USARL.

References

1. COPRTHR SDK (2012), `http://www.browndeertechnology.com/coprthr.htm`
2. HIPAcc - Heterogeneous Image Processing Acceleration (2012), `http://hipacc.sourceforge.net/index.html`
3. Borkar, S.: The exascale challenge. In: 2010 International Symposium on VLSI Design Automation and Test (VLSI-DAT), pp. 2–3 (2010)
4. Brown Deer Technology: STDCL: A Simplified C Interface for OpenCL, revision 1.4 (2012)
5. Calxeda: Calxeda launches the energycore processor; delivers 10 times the performance for the same power (2013), `http://www.calxeda.com`
6. Carroll, A., Heiser, G.: An analysis of power consumption in a smartphone. In: Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, USENIXATC 2010, p. 21. USENIX Association, Berkeley (2010)
7. Fang, J., Varbanescu, A.L., Sips, H.: A comprehensive performance comparison of CUDA and OpenCL. In: Proceedings of the 2011 International Conference on Parallel Processing, ICPP 2011, pp. 216–225. IEEE Computer Society (2011)
8. Furber, S.: ARM System on Chip Architecture. Addison Wesley (2000)
9. Gioiosa, R.: Towards sustainable exascale computing. In: 2010 18th IEEE/IFIP VLSI System on Chip Conference (VLSI-SoC), pp. 270–275 (2010)
10. Jurecko, M., Kocisová, J., Busa Jr., J., Kasanický, T., Domiter, M., Zvada, M.: Evaluation framework for GPU performance based on OpenCL standard. In: ICNC, pp. 256–261 (2010)
11. Komatsu, K., Sato, K., Arai, Y., Koyama, K., Takizawa, H., Kobayashi, H.: Evaluating performance and portability of OpenCL programs. In: The Fifth International Workshop on Automatic Performance Tuning (June 2010)
12. Playne, D.P., Johnson, M.G.B., Hawick, K.A.: Benchmarking GPU devices with N-body simulations. In: Proc. 2009 International Conference on Computer Design (CDES 2009), WorldComp, Las Vegas, USA, July 13-16, pp. 150–156 (2009)
13. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. SIGPLAN Not. 48(6), 519–530 (2013)
14. Shen, J., Fang, J., Sips, H.J., Varbanescu, A.L.: Performance gaps between OpenMP and OpenCL for multi-core CPUs. In: ICPP Workshops, pp. 116–125. IEEE Computer Society (2012)
15. Stone, J.E., Gohara, D., Shi, G.: OpenCL: A parallel programming standard for heterogeneous computing systems. IEEE Des. Test 12(3), 66–73 (2010)
16. Zhang, Y., Sinclair II, M., Chien, A.A.: Improving performance portability in OpenCL programs. In: Kunkel, J.M., Ludwig, T., Meuer, H.W. (eds.) ISC 2013. LNCS, vol. 7905, pp. 136–150. Springer, Heidelberg (2013)