2014 IEEE 28th International Parallel & Distributed Processing Symposium

Energy Efficient HPC on Embedded SoCs: Optimization Techniques for Mali GPU

Ivan Grasso*†, Petar Radojković*, Nikola Rajović*‡, Isaac Gelado*, Alex Ramirez*‡
* *Barcelona Supercomputing Center, Barcelona, Spain*
† *Institute of Computer Science, University of Innsbruck, Austria*
‡ *Universitat Politècnica de Catalunya, Barcelona, Spain*

***Abstract*—A lot of effort from academia and industry has been invested in exploring the suitability of low-power embedded technologies for HPC. Although state-of-the-art embedded systems-on-chip (SoCs) inherently contain GPUs that could be used for HPC, their performance and energy capabilities have never been evaluated. Two reasons contribute to the above. Primarily, embedded GPUs until now, have not supported 64-bit floating point arithmetic – a requirement for HPC. Secondly, embedded GPUs did not provide support for parallel programming languages such as OpenCL and CUDA. However, the situation is changing, and the latest GPUs integrated in embedded SoCs do support 64-bit floating point precision and parallel programming models.**

In this paper, we analyze performance and energy advantages of embedded GPUs for HPC. In particular, we analyze ARM Mali-T604 GPU – the first embedded GPUs with OpenCL Full Profile support. We identify, implement and evaluate software optimization techniques for efficient utilization of the ARM Mali GPU Compute Architecture. Our results show that, HPC benchmarks running on the ARM Mali-T604 GPU integrated into Exynos 5250 SoC, on average, achieve speed-up of 8.7× over a single Cortex-A15 core, while consuming only 32% of the energy. Overall results show that embedded GPUs have performance and energy qualities that make them candidates for future HPC systems.

***Keywords*-High performance computing; Embedded GPUs; Optimization; Performance analysis.**

I. Introduction

The high performance computing (HPC) community has recognized GPUs as powerful parallel computing engines, well suited for computationally demanding applications with extensive data-level parallelism. A broad range of computational algorithms from different HPC domains have been successfully ported to GPUs, and HPC systems that incorporate GPU engines have shown distinct performance and energy-efficiency improvements over their counterparts that are based only on traditional microprocessors.

On the other hand, in order to improve energy efficiency, the community has been deploying large-scale HPC systems from SoCs that are based on embedded processors [1], [2], [3], [4], [5]. Although state-of-the-art embedded SoCs do integrate GPUs that could provide significant improvements in terms of performance and energy efficiency, up to now, the embedded GPUs have not been used for high performance computing. The main reason behind this is that the GPUs integrated in embedded SoCs did not support parallel programming models such as OpenCL or CUDA, and neither did they support 64-bit floating-point precision.

Nowadays the situation is changing, and the latest embedded GPUs [6], [7], [8] satisfy 64-bit floating-point arithmetic precision constraints, and support computational languages such as OpenCL or CUDA. Aware of this upcoming trend, we analyze the use of the ARM Mali GPU Compute Architecture for HPC workloads. Our work explores whether embedded GPUs can provide energy and performance advantages over the embedded CPUs, and emphasizes the importance of OpenCL software optimizations.

Our study makes the following contributions:

- First, we investigated the possibility of using embedded GPUs for high performance computing. We successfully ported nine HPC benchmarks to OpenCL and executed them on ARM Mali-T604 GPU [6] – the first embedded GPU with OpenCL Full Profile support. We evaluated performance, power consumption and energy-to-solution of the benchmarks and compared the results with ARM Cortex-A15 cores.
- Second, we identified the important OpenCL software optimization techniques for efficient utilization of the ARM Mali GPU Compute Architecture. This architecture differs in many aspects from conventional high-end GPU architectures and OpenCL code must be optimized taking into account the particular characteristics of the underlying hardware.
- Finally, we deployed the proposed Mali GPU optimization techniques on nine HPC benchmarks and we evaluated the impact of the optimization techniques on system performance and energy consumption.

Our results show that the Mali-T604 GPU provides distinct improvements in terms of performance and energy-to-solution over ARM Cortex-A15. On average, single-precision and double-precision GPU benchmarks achieve a speedup of 8.7× over the benchmarks running on a single Cortex-A15 core, while consuming only 32% of the energy. Our study confirms that, in high performance computing, embedded GPUs can offer performance and energy advantages over embedded CPUs, similar to their high-end counterparts in heterogeneous supercomputers.

The rest of the paper is organized as follows. Sections II

1530-2075/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPS.2014.24

describes the ARM Mali T-604 GPU Architecture and the OpenCL Programming Model. Section III presents the OpenCL optimization techniques. Section IV describes the experimental environment used in the study. In Section V, we present the results of our experiments. Section VI presents the related work, while Section VII lists the conclusions of the study.

II. Background

A. ARM Mali T-604 GPU Architecture

While the main focus of embedded GPUs is 2D and 3D high quality graphics, embedded GPU companies are also becoming interested in the general purpose compute capabilities of such devices. The first effort in this direction from ARM Holdings is the Mali-T600 GPU Midgard series. The Mali GPU Compute Architecture is designed to fully comply with the OpenCL Full Profile which has strict requirements for precision and support for mathematical functions. The first GPU based on this architecture is the Mali-T604 [6]. In addition to satisfying IEEE-754-2008 precision requirements for single and double-precision floating point, Mali-T604 natively supports 64-bit integer data types and implements barriers and atomics in hardware.

Figure 1 depicts the architectural details of the ARM Mali-T604 GPU. The GPU supports up to four shader cores with two arithmetic pipes per core. The *Job Manager*, implemented in hardware, abstracts the GPU core configuration from the driver and distributes the computational tasks to maximize the GPU resource utilization. The *Memory Management Unit* can easily map memory from the CPU's address space into the GPU's address space to quickly copy or share information. The level 2 cache is shared between the shader cores and maintained coherent by the *Snoop Control Unit*.

B. OpenCL Programming Model

OpenCL [9] is an open industry standard for programming heterogeneous systems composed of devices with different capabilities such as CPUs, GPUs and other accelerators. The platform model comprises of a host connected to one or more compute devices. Each device logically consists of one or more compute units (CUs) which are further divided into processing elements (PEs). Within a program, the computation is expressed through the use of special functions called kernels, that are, for portability reason, compiled at runtime by an OpenCL driver. A kernel represents a data-parallel task and describes the computation performed by a single thread, which is in OpenCL called *work-item*. During the program execution, based on a index space (N-Dimensional Range) a certain number of work-items are generated and executed in parallel. The index space can also be divided into work-groups, each of them consisting of many work-items. The exchange of data between the host and the compute devices is implemented through memory buffers, which are passed as arguments to the kernel before its execution.

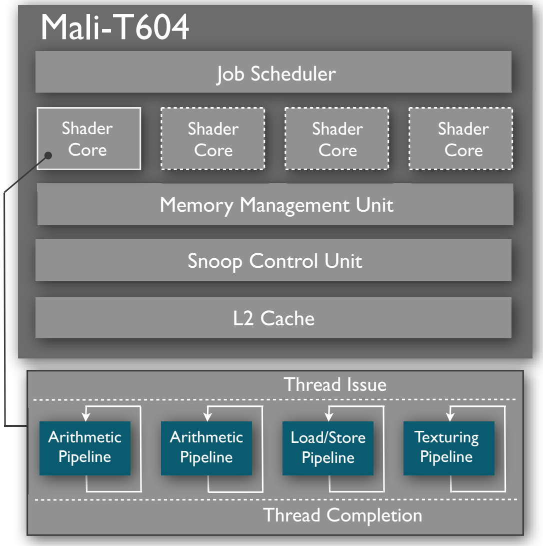


Figure 1. ARM Mali-T604 GPU Architecture

In the past few years, OpenCL has quickly emerged as the *de facto* standard for heterogeneous computing, with the support of many vendors such as Adapteva, Altera, AMD, ARM, Intel, Imagination Technologies, NVIDIA, Qualcomm, Vivante and Xilinx. Recently the OpenCL 2.0 provisional specification has been released [10], to receive feedback before the specification finalization. Most of the desktop and server devices support OpenCL 1.1 or 1.2 Full Profile. The OpenCL 1.1 Embedded Profile is defined as a subset of OpenCL rather than a separate specification for embedded platforms. Most of the restrictions concern 64-bit integer support, 2D and 3D image support and floating point requirements. Although this list will change with the 2.0 specification, the relaxation of the floating point requirement will still be part of the Embedded Profile. Therefore, devices that can be profitably used in a HPC scenario will still have to support the OpenCL Full Profile.

III. OpenCL Optimizations for Mali GPU

OpenCL is a well designed language that allows access to the compute power of heterogeneous devices from a single multi-platform source code. Although the portability at code level ensures the execution of programs in all devices that support the language, the same portability is not present from a performance perspective [11]. Currently it is possible to achieve high performance for a specific device, only through manual code optimizations and tuning. Recently, an interesting investigation in this area has been done by Phothilimthana et al. [12] which proposed an empirical auto-tuning framework for diverse heterogeneous systems able to automatically generate OpenCL code from an implicitly parallel language. Even though early steps in this direction look promising, the OpenCL performance portability is still an ongoing research topic. In this section we analyze and explain the OpenCL software optimization techniques that enable the ARM compiler to efficiently map the program to the specific ARM Mali GPU Compute Architecture.

A. Host Code Optimizations

Memory allocation and mapping. OpenCL is typically executed in desktop and server systems where the application processor and the GPU have separate memories and copy operations have to be used to exchange data between the host and the device. Differently, the Mali GPU Architecture provides a unified memory system where traditional copy operations are not required. Although, from a hardware point of view, time and power consuming data transfers can be avoided, the Mali GPU cannot access memory buffers created with the standard *malloc* function because they are not mapped into the GPU memory space. To allow the GPU to access the data, OpenCL buffer objects have to be created using the *clCreateBuffer* function with the *CL_MEM_USE_HOST_PTR* flag and data has to be copied by the host processor with the *clEnqueueWriteBuffer* and *clEnqueueReadBuffer* OpenCL functions. Although this method makes possible data exchange between host and device, it does not solve the additional copy issue. To eliminate all the computationally expensive copies it is necessary to make the initial memory allocation through the OpenCL API and to use memory mapping functions. Buffers have to be allocated using the *clCreateBuffer* function with the *CL_MEM_ALLOC_HOST_PTR* flag and the *clEnqueueMapBuffer* and *clEnqueueUnmapMemObject* have to be used to enable both the application processor and the Mali GPU to access the data.

Load distribution. Another important aspect of an OpenCL program is the load distribution between the processing elements of a device. The API offers a function called *clEnqueueNDRangeKernel* to execute a kernel on a device. To allow control over the distribution some arguments of the function describe the dimensionality of the data, the number of work-items to be processed for each dimension (*global_work_size*) and the number of work-items in a work-group for each dimension (*local_work_size*).

As suggested in the official Mali OpenCL Developer Guide [13] the optimal *global_work_size* can be calculated as the device *maximum_work-group_size* multiplied by the number of shader cores multiplied by a constant. This constant for the Mali-T604 is four or eight. More generally, the *global_work_size* must be in the order of several thousands to maximize the GPU resources utilization and to achieve high performance. With regard to the local work-group size, if the application is not required to share data among work-items, the Developer Guide [13] suggests to set the *local_work_size* parameter to NULL and let the OpenCL driver to determine the most efficient *work-group_size* for the kernel. Although this practice simplifies the selection of the *work-group_size* value, we noticed that, currently, the driver is not always capable of doing a good selection. During our experimental evaluation we noticed some performance degradation and we strongly suggest to manually tune the *local_work_size* parameter.

B. Kernel Code Optimizations

The ARM Mali GPU Architecture, as depicted in Figure 1, differs in many aspect from the high end GPU architectures used in desktops and servers. Under this consideration, the OpenCL kernel code must be optimized taking into account the particular characteristics of the hardware.

Memory Spaces. Defined as a generic language for heterogeneous computing, OpenCL exposes a memory model composed by several memory spaces that every vendor maps differently to the corresponding available hardware. These four address spaces are: (1) *global memory*, the largest capacity memory on the compute device, visible to all work-items; (2) *constant memory*, similar to *global memory* but read-only; (3) *local memory* to share data between work-items in a work-group; (4) *private memory*, to store data for every individual work-item. Usually dedicated GPUs from AMD and NVIDIA present an on-chip memory with much higher bandwidth and lower latency than global GPU memory. The OpenCL implementations offered by the two vendors map the local memory space to the on-chip memory making the exploitation of memory locality at code level one of the most important factors in achieving high performance. Differently, Mali GPUs have a unified memory system where local memory is physically mapped to the global memory. For this reason traditional code locality optimizations are not required, simplifying the coding of optimized OpenCL kernels.

Thread Divergence. Other optimizations usually used in GPU computing are those regarding warp or wavefront execution. Warps or wavefronts represent the smallest executable units of parallelism on NVIDIA or AMD devices (respectively 32 or 64 work-items). This means that if two work-items inside a warp or a wavefront take divergent paths of the control flow, all work-items of the same unit go through both paths. This issue is called thread divergence and can significantly affect the instruction throughput of the computational kernel. The thread divergence problem is not present in the ARM Mali GPU Architecture because the smallest unit of parallelism is a single thread (work-item) that, being independent, cannot diverge.

Vectorization. One of the most important hardware characteristic to take into account during the development of OpenCL code for the ARM Mali GPU Architecture is that the shader cores contain 128-bit wide vector registers. To allow the effective programming of such units, OpenCL offers vector load and store instructions and vector types of different sizes. Vectorization of an OpenCL scalar code can be done by converting the scalar data types (e.g. float) to vector data type (e.g. float4) increasing the number of elements processed by each work-item and consequently reducing the *global_work_size* in the host code. This optimization not only promotes the use of hardware resources, but also allows a reduction of the run-time scheduling overheads due to the decrease in the number of work-groups.

Vector Sizes. OpenCL, as already mentioned, provides vector types of different sizes that the compiler must map to the corresponding hardware vector registers. In case of the ARM Mali-T604, we noticed that, after vectorizing the kernel, the best achievable performance is not bound to a particular vector size but can vary from case to case. Using types wider than the underlying hardware can improve the instruction-level scheduling, but also increase register pressure. For this reason we suggest, whenever the code allows it, to experiment with different vector sizes (e.g. size of 4, 8, 16). It is also important to keep in mind that vector load and store operations access multiple data elements in memory with a single instruction leading to more efficient use of the available bandwidth. For this reason such operations should be also used in kernels that do not take advantage of vector registers, but process each element of the vector array individually.

Loop Unrolling. Another opportunity for vectorization of the kernel code is at loop level. A loop can be unrolled and multiple instructions can be replaced with a single vector instruction which operates on multiple data elements. Although this optimization usually leads to an increase in the performance on relatively long loops, in case the number of iterations is not a perfect multiple of the vector size, the overhead due to the correct handling of the last iterations of the loop has to be considered. The best performance for different kernel codes can be achieved using vectorization and unrolling, taking into consideration that code replication can also lead to performance degradation.

Data Organization. In vector architectures the organization of data is of primary importance. Typically in application code data is packed in an Array of Structures (AOS). If we consider a set of three dimensional points, their representation would be an array in which each element is a structure with x, y, z coordinates. Although this representation is the most natural, it typically executes poorly in vector register and requires significant loop unrolling to improve its efficiency. A more efficient data-packing approach is Structure Of Arrays (SOA). In this representation the data types are the same across the vector. With this approach the list of points would be organized as three arrays containing respectively the x, y and z values that would facilitate the application of vector instructions increasing the code performance.

Directives and Type Qualifiers. Better performance can also be achieved by passing the compiler information through the use of directives and type qualifiers. With function inlining the user can increase the size of basic blocks (sequences of consecutive instructions without branches) and eliminate the overhead associated with the function call. The use of the *const* keyword allows the compiler to make more assumptions and therefore produce significant optimizations. The restrict qualifier, applicable to the arguments of a kernel, enables the compiler to assume that pointers point to different locations helping to limit the effects of pointer aliasing.

IV. Experimental Environment

A. Benchmarks

In order to evaluate the Mali-T604 GPU and OpenCL optimizations techniques presented in Section III, we used a set of nine HPC kernels [1]. The benchmarks used cover a wide range of algorithms employed in HPC applications and stress various architectural features. These benchmarks have been already used in previous studies [14], [15] to evaluate the suitability of embedded platforms for HPC systems. Below, we briefly describe each of the benchmarks.

Sparse Vector-Matrix Multiplication (spvm) benchmark multiplies a vector and a sparse matrix to produce a new vector. It is useful as metric to measure performance in cases of load imbalance.

Vector Operation (vecop) benchmark performs an addition of two vector in an element-by-element basis. Given the memory-bound nature of the kernel, this benchmark stresses the memory bandwidth of the platform under study.

Histogram (hist) benchmark computes the histogram of the values present in a vector using a configurable bucket size. It uses local privatization that requires a reduction stage which can become a bottleneck on highly parallel architectures.

3D Stencil (3dstc) benchmark produces a new 3D volume from an input 3D volume. Each point of the output is a linear combination of the point with the same co-ordinates in the input and the neighboring points on each dimension. This benchmark is useful to evaluate the performance in presence of memory accesses with regular strides.

Reduction (red) benchmark applies the addition operator to produce a single (scalar) output value from an input vector. Reduction is a common operation in many computational kernels and allows to measure the capability of the compute accelerator to adapt from massively parallel computation stages to almost sequential execution.

Atomic Monte-Carlo Dynamics (amcd) benchmark performs a number of independent simulations using the Markov Chain Monte Carlo method. Initial atom coordinates

[1] In the rest of the paper, we refer to the HPC kernels used in the study as "benchmarks". Word "kernels" is used to refer to OpenCL kernels.

are provided and a number of randomly chosen displacements are applied to randomly selected atoms which are accepted or rejected using the Metropolis method.

N-Body (nbody) benchmark takes as input, a list of bodies described with a set of parameters (position, mass, initial velocity) and updates their information after a given simulated time period based on gravitational interference between each body.

2D Convolution (2dcon) benchmark produces a new matrix from an input matrix of the same size. Each point of the output is a linear combination of the point with the same coordinates in the input and the neighboring points. Differently from the 3D stencil computation, neighboring points can include points with the same coordinates as the input point plus/minus an offset in one or two dimensions. This benchmark is useful to evaluate the performance in presence of spatial locality and strided memory accesses.

Dense matrix-matrix Multiplication (dmmm) benchmark performs the multiplication of two dense input matrices. Matrix multiplication is a common computation in many numerical simulations and measures the ability of the compute accelerator to exploit data reuse and compute performance.

B. Benchmark implementation

Each benchmark was implemented in four different versions: *Serial*, *OpenMP*, *OpenCL*, and *OpenCL Opt*. The *Serial* benchmarks were designed to execute on a single core. The *OpenMP* benchmarks, through the use of threads, were designed to execute in parallel on several CPU cores. In our experiments, the *OpenMP* benchmarks we executed on two Cortex-A15 cores. The *Serial* and the *OpenMP* versions have been already developed and used by previous studies [14], [15] that tested the suitability of embedded platforms for high performance computing. These versions do not make use of vector instructions. This is due to the fact that the ARM Cortex-A15 CPU does not incorporate a double-precision SIMD unit and full IEEE-754-2008 floating-point vector support.

The *OpenCL* versions of the benchmarks were developed in the Open Computing Language to allow a parallel execution on the GPU. Recently, data structures and data layout transformations have been proposed to efficiently implement some of the algorithms used in our benchmarks [16], [17], [18]. However, in order to maintain a similar code base for all CPU and GPU implementations, we do not take advantage of them. One of the main goals of the study is to quantify the performance and energy improvements of the proposed OpenCL optimization techniques. In this regard, we enhanced the *OpenCL* benchmarks following ***only*** the guidelines presented in Section III. The new versions of the benchmarks are referred to as *OpenCL Opt*. To avoid rewriting the host code with different data transfer techniques, both, *OpenCL* and *OpenCL Opt* benchmarks make use of the host memory mapping, exploiting the unified memory system offered by the architecture. Due to the problem of data transfer already being well known in the heterogeneous computing field of research [19], we want to place more emphasis on other important optimization aspects during the comparison between the *OpenCL* and *OpenCL Opt* versions of the benchmarks.

C. Hardware Platform and System Software

The experiments were executed on Samsung Exynos 5 Dual Arndale Board [20]. The board comprises the Samsung Exynos 5250 embedded system-on-chip (SoC) and is equipped with 2 GB of DDR3L-1600 memory. The Samsung Exynos 5250 integrates a dual-core ARM Cortex-A15, running at 1.7 GHz with 32 KB of private L1 instruction and data cache, and 1 MB of shared L2 cache. Alongside the CPU cores, the SoC features a four-core ARM Mali-T604 GPU whose architecture has been already described in Section II-A.

The operating system running in the platform is Ubuntu 11.10 with a Linux kernel version 3.0.31. The operating system image includes the driver needed for executing OpenCL programs on the Mali-T604 GPU. The benchmarks were compiled with GCC version 4.6.1 with $-O3$ optimization flag. Even if the CPU floating-point hardware includes the NEON extension, floating-point vector operations are not automatically generated by GCC's auto-vectorization pass and we did not make use of the *-funsafe-math-optimizations* flag to enable them.

D. Methodology

In all the experiments, the problem size for all the four versions of the same benchmark is maintained constant, so that each version has the same amount of work to perform. The measurements were collected only in the parallel regions of the benchmarks, excluding the initialization and finalization phases. We adjusted the number of iterations of the considered regions so that each benchmark runs for long enough to get an accurate energy consumption figure. We repeated each experiment 20 times and we computed the mean value and the standard deviation of the measured performance and power consumption. In all the presented experiments, the standard deviation is negligible, thus we do not report it.

The power consumption of the board was measured with the Yokogawa WT230 power meter. The WT230 power meter offers a sampling frequency of 10 Hz with 0.1% accuracy.

V. Results

As already mentioned in Section IV-B, we executed and analyzed the results for four versions of the benchmarks. *Serial* and *OpenMP* benchmark versions were executed on one and two ARM Cortex-A15 cores, respectively, while

OpenCL and *OpenCL Opt* versions were executed on the ARM Mali-T604 GPU. In this section, we present comparison between performance, power requirements and energy consumption of the different versions of the benchmarks under study.

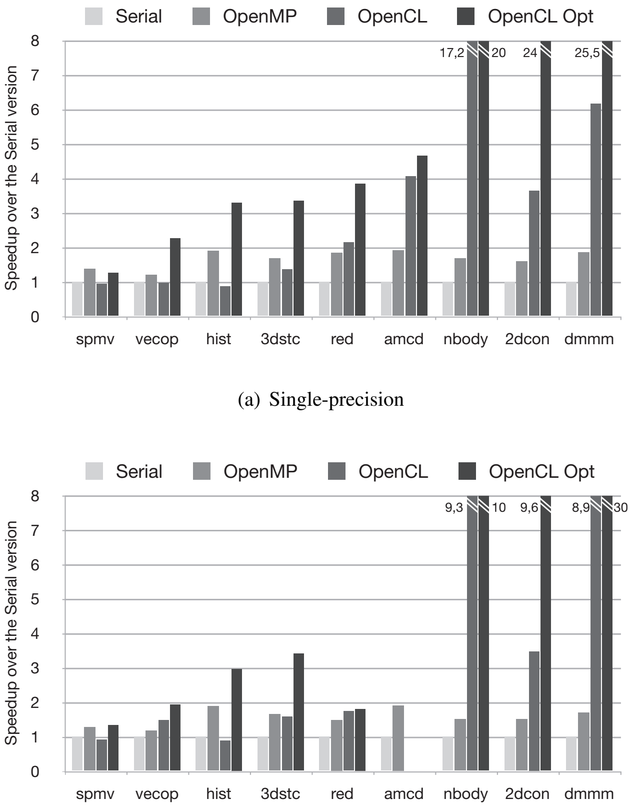
A. Performance Analysis

In Figure 2, we present the performance comparison between different versions of the benchmarks. The benchmarks under study are listed along the X axis of the figure, while the Y axis shows the speedup relative to the *Serial* version of the code. We present the result for two sets of experiments, in single-precision and double-precision.

Single-precision: Single-precision results are presented in Figure 2(a). Speedup of parallelized *OpenMP* benchmarks over the *Serial* version ranges from 1.2× to 1.9×, on average it is 1.7×. Regarding the *OpenCL* version, three out of the nine benchmarks (*spmv*, *vecop*, *hist*) experience performance degradation with respect to the *Serial* code. *OpenCL* version of *3dstc* benchmark experiences a 1.4× speedup over the *Serial* version, however it still experiences lower performance than the *OpenMP* code. For *red*, *amcd*, and *2dcon* benchmarks, the improvement over the *Serial* version is 2.1×, 4.1×, and 3.6×, respectively, overcoming the *OpenMP* version. Finally, *dmmm* and *nbody* benchmarks experience speedup of 6.2× and 17.2×. From these results, we can conclude that porting code to OpenCL and running on the GPU, on its own, does not guarantee significant performance improvement, and that a multithreaded *OpenMP* version may lead to better performance.

Significantly different results were obtained on applying optimization techniques to the OpenCL code (*OpenCL Opt*). Sparse Vector-Matrix Multiplication (*spmv*) is the only application that does not perform well, but, we still detect some performance improvement over the *Serial* implementation (1.25× speedup). Four of the nine benchmarks (*vecop*, *hist*, *3dstc*, *red*) show a speedup between 2× and 4×. Benchmark *amcd* experiences speedup of 4.7×, while benchmarks *nbody*, *2dcon*, and *dmmm* show improvement of 20×, 24× and 25.5×, respectively.

Discussion: The significant difference in performance improvement for different *OpenCL* and *OpenCL Opt* benchmarks can be explained by analyzing the benchmarks' characteristics. Each of the benchmarks under study requires a certain amount of data from external memory. Therefore, in absence of sufficient computation, the memory bandwidth can limit the performance. Sparse Vector-Matrix Multiplication (*spmv*) and Vector Operation (*vecop*) with large working sets and little computation are good examples of this scenario. As already mentioned in Section IV-B, our OpenCL versions do not take advantage of special data structures and for this reason, *spmv* can only partially exploit the available bandwidth.



(a) Single-precision

(b) Double-precision

Figure 2. Performance results for the benchmarks executed on the Samsung Exynos 5250 SoC

Histogram (*hist*) makes use of atomic operations supported at hardware level to compute the result and shows good performance compared to the *Serial* implementation. 3d Stencil (*3dstc*) does not take advantage of vector instruction and limits the optimizations to work-group size tuning and data reuse. Reduction (*red*) makes use of a two-stage reduction, that performs a constant number of parallel reductions based on the number of used work-groups. The main difference in performance between *OpenCL* and *OpenCL Opt* for this benchmark is due to the vectorization and the use of a tuned work-group size. The *OpenCL* version of *amcd* without optimizations can already reach a speedup of 4.1×. We did not find many hot spots for optimizations and the *OpenCL Opt* is only slightly faster.

The last three applications can reach significant speedups (up to 25.5×) over the *Serial* implementations. The *OpenCL* version of N-Body (*nbody*) does not apply any change to the main data structure representation that would lead to an easier applicability of vector optimizations.

For this reason, the *OpenCL Opt* version does not show significant improvements over the non-optimized version. Differently, 2D Convolution (*2dcon*) and Dense matrix-matrix Multiplication (*dmmm*) provide extensive parallelism at both vector and thread level. In these cases most of the optimizations can be successfully applied (loop unrolling, vectorization, group-size and vector-size tuning) leading to a considerable increase in performance.

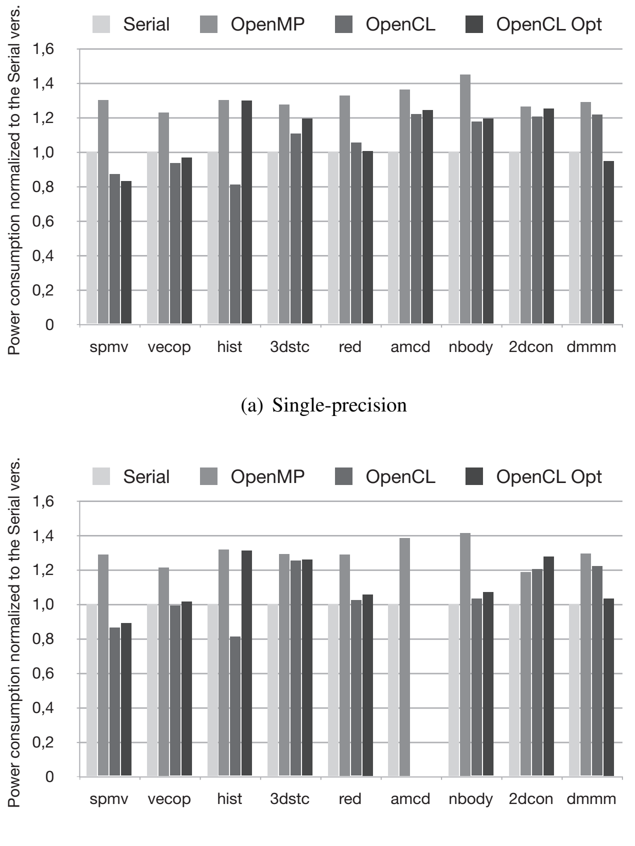
Double-precision: Double-precision results are presented in Figure 2(b). The Atomic Monte-Carlo Dynamics (*amcd*) *OpenCL* versions are not presented due to a compiler issue that does not allow the correct termination of the compilation phase for the *OpenCL* kernel in double precision. Since the compiler source code is not publicly available, we could not solve the problem. The problem is reported, and it will be corrected in a future version of the compiler. For the remaining benchmarks, we detect similar trends as for single-precision.

For the *OpenCL* version, two of the eight benchmarks (*spmv*, *hist*) show lower performance than the *Serial* version. *OpenCL* version of *3dstc* benchmark experiences a 1.6× speedup over the *Serial* version, however it still experiences lower performance than the *OpenMP* code. Benchmarks *vecop* and *red* show 1.5× and 1.7× speedup over the *Serial* version of the benchmarks, and negligible performance improvement over the *OpenMP* code. Finally, three benchmarks experience a significant speedup: *2dcon* (3.5×), *dmmm* (8.9×) and *nbody* (9.3×). Regarding the *OpenCL Opt* version, three of the eight benchmarks (*spmv*, *vecop*, *red*) show a performance improvement below 2×. Benchmarks *hist* and *3dstc*, experience a speedup of 3× and 3.4×. Finally, *2dcon*, *nbody*, *dmmm* benchmarks show speedup is 9.6×, 10×, and 30×, respectively.

Discussion: The speedup of *OpenCL Opt* over the *OpenCL* benchmarks in single-precision (Figure 2(a)) and double-precision (Figure 2(b)) is comparable. However, for the *red*, *nbody* and *2dcon* benchmarks, we have detected a significantly different trend. The reduction of the performance gap between the *OpenCL* and *OpenCL Opt* version of benchmarks *nbody* and *2dcon* in double-precision is due the failure of optimized version of the kernels (with *CL_OUT_OF_RESOURCES* OpenCL Error). The benchmarks were executed without the error in single-precision. The performance loss of *OpenCL Opt red* benchmark in double-precision compared to single-precision is under investigation.

B. Power Analysis

In Figure 3, we present the comparison between power consumption of *Serial*, *OpenMP*, *OpenCL* and *OpenCL Opt* versions of the benchmarks. The benchmarks under study are listed along the X axis of the figure, while the Y axis shows the power consumption relative to the *Serial* version of the code. We present the result for two sets of experiments, in single-precision and double-precision.



(a) Single-precision

(b) Double-precision

Figure 3. Power consumption results for the benchmarks executed on the Samsung Exynos 5250 SoC

Single-precision: Single-precision results are presented in Figure 3(a). Increase in power consumption of *OpenMP* benchmarks over the *Serial* version ranges from 23% (*vecop*) to 45% (*nbody*), and on average is 31%. Results vary insignificantly between *OpenCL* and *Serial* versions of the benchmarks. For *spmv*, *vecop*, and *hist*, the power consumption of the *OpenCL* version is 13%, 7%, and 19% decreased with respect to the *Serial* version. The remaining *OpenCL* benchmarks have a slightly higher power consumption of up to 22% (*amcd* and *dmmm* benchmarks). On average, *OpenCL* benchmarks show only 7% higher power consumption than the *Serial* version. Seven out of nine *OpenCL Opt* benchmarks have power consumption that is very similar to the consumption of the corresponding *OpenCL* benchmarks. Significant difference is detected only for *hist* benchmark that experienced significant power increase with respect to *OpenCL* version, and *dmmm* benchmark that showed significant power reduction. The reasons for the difference in power consumption between *OpenCL* and *OpenCL Opt* versions of the *hist* and *dmmm* benchmarks are to be explored.

Double-precision: Double-precision results are presented in Figure 3(b). The results follow similar trends as the single-precision results.

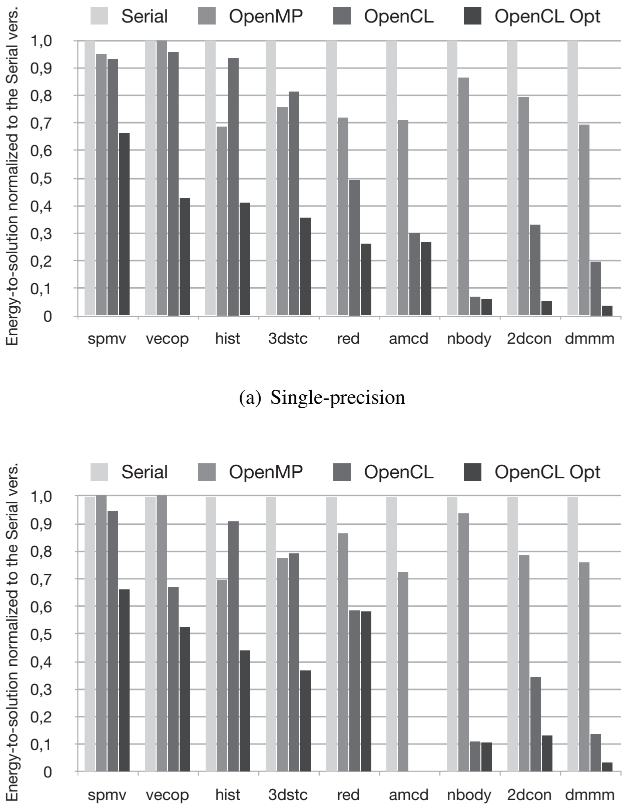
C. Energy-to-Solution Analysis

Figure 4 shows the energy-to-solution of the benchmarks in single-precision and double-precision. The results presented are normalized with respect to the energy-to-solution of the *Serial* implementation of the benchmarks.

Single-precision: Single-precision results are shown in Figure 4(a). Energy-to-solution reduction of *OpenMP* benchmarks over the *Serial* version is 20% on average. The results for *OpenCL* versions of the benchmarks can be clustered in two groups. Benchmarks *spmv*, *vecop*, *hist* and *3dstc* experience moderate energy-to-solution improvement (less than 20%) over the *Serial* code. Actually, *OpenCL* versions of *hist* and *3dstc* consume more energy than corresponding *OpenMP* implementations. For the remaining five *OpenCL* benchmarks (*red*, *amcd*, *nbody*, *2dcon*, *dmmm*), energy-to-solution improvement (reduction) is significant, and it ranges from 51% (*red*) to 93% (*nbody*). Finally, for all the benchmarks under study, *OpenCL Opt* versions experience the lowest energy-to-solution. Improvement with respect to *Serial* version of the benchmarks ranges from 34% (*spmv*) to 96% (*dmmm*), and on average it is 72%.

For all the benchmarks under study, *OpenCL Opt* benchmarks have better energy-to-solution than the corresponding non-optimized *OpenCL* implementations. For seven out of nine benchmarks (*spmv*, *vecop*, *hist*, *3dstc*, *red*, *2dcon*, and *dmmm*), the energy consumption improvement due to proposed OpenCL optimization is significant. On average, the *OpenCL Opt* benchmarks require only 28% of the energy consumed by the *Serial* implementation, while non-optimized *OpenCL* implementations require 56%.

Double-precision: Double-precision energy-to-solution results are presented in Figure 4(b). The results follow similar trends as the single-precision results. The only significant difference is detected for *red OpenCL Opt* benchmark. In this case, energy-to-solution of double-precision increased significantly with respect to the single-precision version. This is due to the relative performance loss of *OpenCL Opt red* benchmark in double-precision with respect to single-precision (see Figure 2). On average, the *OpenCL Opt* benchmarks require only 36% of the energy consumed by the *Serial* implementation, while non-optimized *OpenCL* implementations require 56%.



(a) Single-precision

(b) Double-precision

Figure 4. Energy-to-solution results for the benchmarks executed on the Samsung Exynos 5250 SoC

D. Results Summary

In this section, we analyzed performance, power consumption and energy-to-solution of four different implementations of the benchmarks. *Serial* and *OpenMP* benchmarks were executed on one and two ARM Cortex-A15 cores, respectively, while *OpenCL* and *OpenCL Opt* implementations were executed on the Mali-T604 GPU.

It is important to emphasize that, despite much higher theoretical peak performance, the ARM Mali-T604 GPU usually cannot be effectively exploited with the creation of a simple OpenCL parallel version from the serial code.

Based on the presented results, we see that achieving high performance on an ARM Mali-T604 GPU requires proper utilization of the underlying architecture that can only be obtained through the use of code optimizations and parameter tuning. OpenCL optimization techniques presented in this paper leaded to significant performance improvement for most of the benchmarks under study. On the other hand, we showed that power consumption varies insignificantly between optimized and non-optimized versions of the OpenCL benchmarks. Significant performance improvement on the one hand, and similar power consumption on the other, translate into significantly lower (better) energy-to-solution of the optimized OpenCL benchmarks. Finally, *OpenCL Opt* benchmarks, implemented following the guidelines presented in this paper, show distinct advantage in terms of performance and energy-to-solution over the *Serial* and *OpenMP* benchmark implementations. On average, single-precision and double-precision *OpenCL Opt* benchmarks achieve a speedup of 8.7× over the corresponding *Serial* benchmarks running on the Cortex-A15 core, while consuming only 32% of the energy. Overall, the results clearly show that embedded GPUs have performance and energy qualities that make them good candidates for next generation HPC systems.

VI. Related Work

Recently, embedded GPU computing has drawn a lot of attention from the research community and the industry. In the embedded world, the primary graphics programming interface is OpenGL ES [21]. OpenGL ES is a subset of the well known OpenGL 3D graphics application programming interface (API). Unlike the the standard used in desktop systems and gaming consoles the Embedded Systems version removes some of the functionalities and at the same time extends it to better support the computational abilities of embedded processors. Although the OpenGL ES API is mainly designed for graphics, some recent studies have successfully investigated the acceleration of algorithms in embedded devices.

Singhal et al. [22], [23] explored the implementation, optimization and evaluation of image processing and computer vision algorithms. Cheng et al. [24] investigated the computing power and energy consumption of a embedded CPU-GPU platform for computer vision applications. Lopez et al. [25] presented the first embedded GPU implementation of Local Binary Pattern feature extraction. Rister et al. [26] implemented an efficient implementation of the Scale-Invariant Feature Transform (SIFT) feature detection algorithm.

Due to the difficulty in mapping algorithms to graphics operations and accessing the unexposed low-level hardware characteristics, all the previously mentioned works are specific to the field of image analysis and processing, leaving out a wide range of possible applications.

With the growth of the Open Computing Language (OpenCL) [9] as an open standard for general-purpose parallel programming of heterogeneous server and desktop systems, also the embedded community has become interested in simplified model for embedded GPU computing.

Leskelä et al. [27] created a programming environment based on the standard OpenCL Embedded Profile and tested their embedded CPU-GPU back-ends implementation against an image processing workload. Wang et al. [28] studied the performance of a exemplar-based inpainting algorithm on the Qualcomm Snapdragon S4 chipset [29] which supports the OpenCL Embedded Profile for both CPU and GPU. Although these studies are using OpenCL, they are still restricted to the image processing area and do not investigate whether non-graphics workloads may benefit from embedded GPUs usage.

Other works have investigated how to simulate and evaluate GPUs present in embedded devices. TEAPOT [30] provides full-system cycle-accurate GPU simulation and power model for embedded workloads. The system has been used to analyze techniques that trade image quality for energy saving [30], exploit similarity between consecutive frames to save memory bandwidth [31] and hide memory latency using a decoupled access/execute paradigm [32].

The work that is closest to ours is that of Maghazeh et al. [33]. The authors investigated if a heterogeneous embedded platform with GPUs and CPUs can be an attractive solution for different kinds of applications. They implemented five benchmarks from different application domains and performed experiments on an i.MX6 Sabre Lite development board with OpenCL Embedded Profile support.

Following a similar approach, our work is different in several ways. First, our analysis is focused on high performance computing. This is an important difference as our investigation was enabled by devices with OpenCL Full Profile support wich include double-precision FP64 and full IEEE-754-2008 floating-point. These features are needed for scientific computing workloads and for this reason the ARM Mali-T604 is the first embedded GPU that can be considered in a HPC scenario. Moreover, we present and analyze the performance, power, and energy comparison between the benchmarks in single and double-precision arithmetic. Finally, being aware of the OpenCL performance portability issue, we identified and evaluated software optimization techniques for efficient utilization of the ARM Mali GPU Compute Architecture.

VII. Conclusion

In order to improve energy efficiency, academia and industry have been studying the suitability of low-power embedded technologies for high performance computing. Although state-of-the art embedded SoCs include GPUs that could deliver significant performance and energy improvements, until now, the HPC capabilities of such components have not been explored.

In this paper, we analyzed the suitability of the ARM Mali GPU Compute Architecture for HPC workloads. We successfully ported nine HPC benchmarks to OpenCL and executed them on the ARM Mali-T604 GPU – the first embedded GPU with OpenCL Full Profile Support. Also, we identified and evaluated important OpenCL software optimization techniques for efficient utilization of the ARM Mali GPU Compute Architecture. Our results show that the Mali-T604 GPU provides distinct performance and energy improvements over ARM Cortex-A15 cores, achievable through the use of code optimizations and parameter tuning, taking into account the particular characteristics of the underlying architecture.

Overall, our study confirms that in high performance computing, embedded GPUs can offer performance and energy advantages over embedded CPUs, making them promising candidates for next generation HPC systems.

Acknowledgment

This research has been supported by the Mont-Blanc project (European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement n^o 288777 and 610402), the Spanish Ministry of Science and Technology through *Computacion de Altas Prestaciones (CICYT) VI* (TIN2012-34557), the Spanish Government through *Programa Severo Ochoa* (SEV-2011-0067) and by the FWF Doctoral School CIM Computational Interdisciplinary Modelling under contract W01227.

References

[1] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas, "Overview of the Blue Gene/L system architecture," *IBM Journal of Research and Development*, vol. 49, no. 2, 2005.

[2] IBM Blue Gene Team, "Overview of the IBM Blue Gene/P project," *IBM Journal of Research and Development*, vol. 52, no. 1/2, 2008.

[3] ——, "Blue Gene/Q: by co-design," *Journal of Computer Science - Research and Development*, vol. 28, no. 2-3, 2013.

[4] "Mont-Blanc (FP7-ICT-2011-7 European project): European scalable and power efficient HPC platform based on low-power embedded technology." 2011.

[5] "Mont-Blanc 2 (FP7-ICT-2013-10 European project), European scalable and power efficient HPC platform based on low-power embedded technology." 2013.

[6] "Mali-T604 GPU Architecture," http://www.arm.com/products/multimedia/mali-graphics-plus-gpu-compute/mali-t604.php, 2013.

[7] "Tegra K1," http://www.nvidia.com/object/tegra-k1-processor.html, 2014.

[8] "PowerVR Graphics," http://www.imgtec.com/powervr/powervr-graphics.asp, 2013.

[9] Khronos OpenCL Working Group, "The OpenCL 1.2 specification," http://www.khronos.org/opencl, 2012.

[10] ——, "The OpenCL Specification. Version 2.0," http://www.khronos.org/registry/cl/specs/opencl-2.0.pdf, 2013.

[11] P. Thoman, K. Kofler, H. Studt, J. Thomson, and T. Fahringer, "Automatic opencl device characterization: guiding optimized kernel design," in *Euro-Par*, 2011, pp. 438–452.

[12] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. P. Amarasinghe, "Portable performance on heterogeneous architectures," in *ASPLOS*, 2013, pp. 431–444.

[13] "Mali-T600 Series GPU OpenCL Developer Guide," http://malideveloper.arm.com/, 2013.

[14] N. Rajovic, P. Carpenter, I. Gelado, N. Puzovic, A. Ramirez, and M. Valero, " Supercomputing with Commodity CPUs: Are Mobile SoCs Ready for HPC? ," in *SC*, 2013.

[15] N. Rajovic, A. Rico, J. Vipond, I. Gelado, N. Puzovic, and A. Ramirez, " Experiences with mobile processors for energy efficient HPC ," in *DATE*, 2013, pp. 464–468.

[16] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Corporation, Technical Report NVR-2008-004, 2008.

[17] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *ICS*, 2013, pp. 273–282.

[18] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan, "Data Layout Transformation for Stencil Computations on Short-Vector SIMD Architectures," in *CC*, 2011, pp. 225–245.

[19] C. Gregg and K. Hazelwood, "Where is the data? Why you cannot debate CPU vs. GPU performance without the answer," in *ISPASS*, 2011, pp. 134–144.

[20] "Samsung Exynos 5 Dual Arndale Board," http://www.arndaleboard.org/, 2013.

[21] Khronos OpenGL Working Group, "The OpenGL ES 3.0 specification," http://www.khronos.org/opengles/, 2013.

[22] N. Singhal, I. K. Park, and S. Cho, "Implementation and Optimization of Image Processing Algorithms on Handheld GPU," in *ICIP*, 2010.

[23] N. Singhal, J. W. Yoo, H. Y. Choi, and I. K. Park, "Implementation and Optimization of Image Processing Algorithms on Embedded GPU," in *IEICE TRANSACTIONS on Information and Systems*, vol. 95, no. 5, 2012.

[24] K.-T. T. Cheng and Y.-C. Wang, "Using Mobile GPU for General-Purpose Computing A Case Study of Face Recognition on Smartphones," in *VLSI-DAT*, 2011.

[25] M. B. Lopez, H. Nykanen, J. Hannuksela, O. Silven, and M. Vehvilainen, " Accelerating image recognition on mobile devices using GPGPU ," in *Proceedings of SPIE*, vol. 7872, 2011.

[26] B. Rister, G. Wang, M. Wu, and J. R. Cavallaro, "A fast and efficient sift detector using the mobile GPU," in *ICASSP*, 2013.

[27] J. Leskelä, J. Nikula, and M. Salmela, "OpenCL embedded profile prototype in mobile device," in *SiPS*, 2009, pp. 279–284.

[28] G. Wang, Y. Xiong, J. Yun, and J. R. Cavallaro, "Accelerating Computer Vision Algorithms Using OpenCL on the Mobile GPU A Case Study," in *ICASSP*, 2013.

[29] "Qualcomm Snapdragon Processor," http://www.qualcomm.com/chipsets/snapdragon, 2013.

[30] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "TEAPOT: a toolset for evaluating performance, power and image quality on mobile graphics systems," in *ICS*, 2013, pp. 37–46.

[31] ——, "Parallel Frame Rendering: Trading Responsiveness for Energy on a Mobile GPU," in *PACT*, 2013.

[32] ——, "Boosting mobile GPU performance with a decoupled access/execute fragment processor," in *ISCA*, 2012, pp. 84–93.

[33] A. Maghazeh, U. D. Bordoloi, P. Eles, and Z. Peng, "General Purpose Computing on Low-Power Embedded GPUs: Has It Come of Age?" in *SAMOS*, 2013.