General Purpose Computing on Low-Power Embedded GPUs: Has It Come of Age?

Arian Maghazeh Unmesh D. Bordoloi Petru Eles Zebo Peng
Department of Computer and Information Science, Linköpings Universitet, Sweden
E-mail: {arian.maghazeh, unmesh.bordoloi, petru.eles, zebo.peng}@liu.se

***Abstract*— In this paper we evaluate the promise held by low-power GPUs for non-graphic workloads that arise in embedded systems. Towards this, we map and implement 5 benchmarks, that find utility in very different application domains, to an embedded GPU. Our results show that apart from accelerated performance, embedded GPUs are promising also because of their energy efficiency which is an important design goal for battery-driven mobile devices. We show that adopting the same optimization strategies as those used for programming high-end GPUs might lead to worse performance on embedded GPUs. This is due to restricted features of embedded GPUs, such as, limited or no user-defined memory, small instruction-set, limited number of registers, among others. We propose techniques to overcome such challenges, e.g., by distributing the workload between GPUs and multi-core CPUs, similar to the spirit of heterogeneous computation.**

I. Introduction

Over the span of last decade, researchers have re-engineered sequential algorithms from a wide spectrum of applications in order to harness the inherent parallelism offered by GPUs (Graphics Processing Units) and demonstrated tremendous performance benefits [14]. In fact, brisk successes from early endeavors meant that GPGPU (General Purpose computing on Graphics Processing Units) was established as a specialization on its own. It is not uncommon, nowadays, to find commercial GPGPU applications in electronic design, scientific computing and defense, among others. As GPGPU research has matured, the emerging trend for future is “heterogeneous computation”, where multi-cores, GPUs and other units are utilized synergistically to accelerate computationally expensive problems. Heterogeneous computation is well-poised to become the defacto computational paradigm in the realm of servers and desktops [3].

Yet, in embedded devices, the role of GPUs has been limited, until recently. Over the last 18 months, programmable GPUs have penetrated embedded platforms providing graphics software designers with a powerful programmable engine. Typically, such embedded GPUs are programmed with OpenGL, Renderscript and other similar tools that require prior experience in graphics programming and, in some cases, even expertise on the underlying graphics pipeline hardware [14]. However, today, the stage seems to be set for a change as several industrial players, either, already have, or are on the verge of releasing embedded platforms with low-power GPUs that are programmable by OpenCL. OpenCL is a framework for coding that enables seamless programming across heterogeneous units including multi-cores, GPUs and other processors. Vivante’s embedded graphics cores [17], ARM’s Mali [12] graphics processors, the StemCell Processor [18] from ZiiLabs (Creative) are some examples of low-power embedded GPUs targeted for mobile devices.

Our contributions: We believe that the arrival of OpenCL-enabled GPUs gives us an opportunity to usher in the era of heterogeneous computation for embedded devices as well. However, unless GPGPU on *low-power embedded* platforms can be unleashed, heterogeneous computation will remain debilitated. The question has remained open whether (and what kind of) non-graphics workloads may benefit from embedded GPUs given that powerful multi-core CPUs on the same chip are already a promising choice. Towards this, we mapped and implemented 5 benchmarks — Rijndael algorithm, bitcount, genetic programming, convolution and pattern matching — to an embedded GPU from Vivante. These algorithms are deployed in numerous potential applications including automotive (bitcount, convolution, security), radar or flight tracking (pattern matching) and image processing/augmented reality (convolution). Our choice was also driven by a desire to investigate whether computationally heavy optimization algorithms (genetic programming), typically not suitable on embedded platforms, may become feasible with the advent of low power GPUs. Our results show that embedded GPUs are indeed, sometimes, attractive alternatives for non-graphics computation but, at the same time, intelligent tradeoffs/choices must be considered for each application between its GPU, multi-core and hybrid (heterogeneous) implementations. To the best of our knowledge, ours is the first paper to implement such non-graphic workloads on *embedded* GPUs and compare against sequential and multi-core implementations.

In the last 10 years, several hundred papers have been published on GPGPU but they were almost exclusively concerned about high-end GPUs where maximizing the speedup has been the sole overarching aim. Our experiments reveal that adopting the same optimization strategies as those used for high-performance GPU computing might lead to worse performance on embedded GPUs. This is due to restricted features of embedded GPUs like limited or no user-defined memory, limited size of cache, among several others. Moreover, embedded GPUs share the same physical memory with the CPUs, which implies higher contention for memory, imposing new demands on programmers. We discuss techniques, such as distribution of the workload between GPUs and multi-core CPUs, similar to the spirit of heterogeneous computation to overcome some of the challenges.

II. Related Work

Major strides have been made in GPGPU [14] programming over the years. Almost all threads of work, however, singularly focused on improving performance without discussing other concerns that arise specifically in embedded GPUs. Hence, the existing body of work does not provide any insight into opportunities and challenges of embedded GPGPU programming. Of late, few initial attempts have been made to bridge this gap, however, almost all of them evaluated their work on high-end GPUs.

In a recent paper, Gupta and Owens [4], discussed strategies for memory optimizations for a speech recognition application. However, they did not discuss the impact of their algorithm on power or energy consumption. In fact, they evaluated their performance results on a 9400M Nvidia GPU which is not targeted towards low-power devices such as hand-held smart phones. Yet, we refer the interested reader to their paper for an excellent discussion on limitations that arise out of (i) on-chip memory sharing between GPU and CPU and (ii) limited or no L2 cache. These two characteristics are among several restrictions in embedded GPUs that we target in this paper.

We also note that Mu et al. [13] implemented the benchmarks from High Performance Embedded Computing Challenge Benchmark from MIT [15] on a GPU. However, their paper suffers from two significant drawbacks. First and foremost, they did not study power or energy consumption which is crucial for embedded devices. In fact, all the experiments were performed on the power-hungry Fermi machine from Nvidia. Second, their reported speedups do not include the overheads of data transfer between the CPU and GPU.

There has been some prior work [1], [6], [11], [16] related to modeling power and energy consumption on GPUs. Again, this thread of work has focused on high performance GPU computing and desktop/server workloads, shedding no light on their suitability for low-power applications.

Researchers from Nokia published results [10] that they obtained from OpenCL programming of an embedded GPU. Unfortunately, this work was limited to an image processing algorithm and hence, unlike our paper, it does not provide insight into the applicability of GPUs for any wider range of non-graphic workloads. In contrast to them, we do not restrict our evaluation to one compute intensive GPU algorithm. Rather, we study a set of applications, with varying characteristics, from a wide range of application domains. Also, they did not discuss the differences in optimization strategies between embedded and high-end GPUs while we include a comparative study with results obtained from implementing the same algorithms on a high-end GPU.

III. Methodology

This section describes our benchmarks, the hardware (GPU) platform, operating system, programming languages, and measurement methodologies.

A. Choice of Benchmarks

GPUs have been architected to be amenable for programs that present high degree of data parallelism. Unlike CPUs, GPUs devote less resources (like cache hierarchies) to hide memory latencies and put more emphasis on computational resources. Thus, applications with higher ratios of compute-to-memory access are known to extract higher performance benefits from GPUs. However, unlike graphics programs, non-graphics applications might not exhibit this property even if they are computationally intensive. As such, our primary consideration for choosing the benchmarks was to ensure that some benchmarks have a high ratio of compute-to-memory access while others have a low ratio of compute-to-memory access. Second, we believe that the benchmarks should cover a wide application range. As mentioned in Section I, the 5 benchmarks that we implemented may be utilized in automotive software, radar/flight tracking, image processing, augmented reality and optimization tools. We hope that our work will spur the embedded systems community to explore other potential applications on embedded GPUs. With these methodological choices in mind, we chose 5 benchmarks. The characteristics for each benchmark is individually discussed below.

We chose the **Rijndael** algorithm [2] that is specified in the Advanced Encryption Standard. Data security is increasingly becoming more important as mobile and hand-held devices are beginning to host e-commerce activities. In fact, it is also gaining significance in vehicular systems as the internet continues to penetrate the automotive applications. Moreover, the characteristics of the version we chose to implement inherently makes it an excellent candidate to stress and test the memory bandwidth of the GPU because the algorithm relies heavily on look-up tables. It essentially has a low compute-to-memory access ratio.

Second, we selected the **bitcount** application from the Automotive and Industrial Control suite of MiBench [5]. The *bitcount* is an important benchmark because low-level embedded control applications often require bit manipulation and basic math functionalities. It counts the total number of bits set to 1 in an array of integers. Thus, it needs to sum several integers, thereby stressing the integer datapath of the GPU. The bit count algorithm is also interesting because it involves two typical GPGPU algorithmic paradigms (i) "synchronization" across the threads in the GPU and (ii) "reduction" in order to perform the summation. Thus, it is an excellent example to study the impact of conventional GPGPU programming tricks on embedded GPUs. **Bitcount** has a low compute-to-memory access ratio.

Third, we selected a **genetic programming** algorithm [8] and this choice was driven by a desire to investigate whether computationally heavy optimization algorithms (genetic programming), typically not suitable on embedded platforms, may become feasible with the advent of low power GPUs. We chose the problem of Intertwined Spirals to be solved with genetic programming because it is a classic classification problem and has been extensively studied [7]. The chosen algorithm is very interesting because it also involves a "reduction" component apart from floating point operations. This application has a very high compute-to-memory access ratio.

We also selected **convolution**, a widely used image processing algorithm, for implementation on the GPU because optimization of convolution on GPU using CUDA and OpenCL has been extensively studied [3]. The convolution code has a very high compute-to-memory access ratio. For our implementation, we selected a standard algorithm [3]. The main goal of the convolution algorithm is to modify each pixel in an image based on a *filter* that uses information from adjacent pixels. The filter is essentially the kernel to be implemented on the GPU and is the compute intensive component of the convolution algorithm. Various effects such as sharpening, blurring, edge enhancing, and embossing are produced by altering only the filter. Such image processing functions play a vital role in medical devices, automotive applications, and augmented reality.

Finally, we selected a **pattern matching** algorithm from the High Performance Embedded Computing Challenge Benchmark from MIT [15]. This algorithm is used by the Feature-Aided Tracker, in radar applications, where the goal is to use high resolution information about the characteristics of a target to aid the identification and tracking of targets. In essence, the pattern matching algorithm considers two vectors of same length and computes a metric that quantifies the degree to which the two vectors match. Apart from their conventional application in avionics, such applications are expected to penetrate intelligent cars very soon. The pattern matching algorithm also has a high compute-to-memory access ratio. However, compared to the other benchmarks, it has a substantially large kernel to be implemented on the GPU. Given the limited memory on GPUs, this makes for another interesting case study.

B. Hardware Platform

All our experiments were performed on the i.MX6 Sabre Lite development board that is targeted for portable computing on battery-powered devices. The board includes four ARM Cortex A9 cores running at 1.2 GHz per core. The chip includes a Vivante GC2000 GPU with 4 SIMD cores, each running at 500MHz. The GPU has a limited hardware cache of only 4KB. Vivante GC2000 has a register size of 2KB in each core (with 4 cores in total). Its instruction cache can accommodate up to 512 instructions in total. Vivante's embedded GPUs are designed for very low power consumption.

It is important to note here that, at the time of the writing of this paper, OpenCL/CUDA drivers for other boards with embedded GPUs are not available in the public realm. As and when such GPU platforms, with their respective software drivers, become openly available, it will be worthwhile to conduct a comparative study between different embedded GPU architectures.

For a comparative study we also implemented our benchmarks on the Nvidia Fermi machine with a Tesla M2050 GPU. It has 14 streaming multi-processors which together have 448 cores running at 1147MHz. The host contains 2 Intel CPUs Xeon E5520, with 8 cores in total and each clocked at 2.27 GHz.

C. Operating System

We utilized the Freescale version of Linux 2.6 package along with a supporting ARM tool-chain.

D. Programming Languages

The GPU kernels were written in OpenCL supported by Vivante's software development kit. The GC2000 GPU, that we study, is supported by OpenCL Embedded Profile 1.1.

OpenCL includes a language (based on C99) for writing kernels (functions that execute on OpenCL compatible devices like GPUs), plus application programming interfaces (APIs) that are used to define and then control the platforms. The use of OpenCL to program GPUs is popular because it has been adopted by Intel, AMD, and Nvidia, among others. It has also been extensively used to program GPUs for non-graphics workload. Finally, OpenCL also provides support for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processing units. In future, as OpenCL drivers for embedded CPU and GPU cores on the same platform become available, OpenCL is likely to lend itself as one of the common frameworks for heterogeneous programming.

E. Measuring the Running Times

The running time reported in this paper, for each of the application, is an average of 5 runs of the application. For each application, we compare running times of its kernels on a single-core, dual-core and quad-core CPU implementations against the GPU implementations. We perform this comparison both on the Fermi (Tesla GPU versus Xeon CPUs) and the Sabre Lite platforms (Vivante GPU versus the ARM CPUs). It should be noted that in some cases, we propose heterogeneous implementations and report results for them. By heterogeneous implementations, we mean that the kernel runs on both the Vivante GPU and the ARM cores (respectively, Tesla GPU and the Xeon CPU).

Broadly, we conducted experiments to investigate the influence of the following architectural and software design decisions. First, note that the Vivante GC2000 GPU does not have hardware support for local memory, i.e., there is no "on-chip" user-defined memory that can be used to share data between GPU cores. However, the OpenCL embedded profile 1.1 mandates software support for local memory in order to enforce consistency of the standard with respect to other GPUs that might, in future, support local memory. We conducted a set of experiments where we compared two implementations — with and without the use of local memory — to understand the implications on performance when a programmer uses local memory in embedded GPUs (like Vivante GC2000) expecting performance benefits as is the practice in conventional GPGPU programming with high-end GPUs.

Second, note that any GPU execution consists of data transfer to the GPU before kernel launch, the execution of the kernel in the GPU and data transfer from the GPU after the kernel execution. As such, we can imagine that there are two major phases (i) kernel execution and (ii) the data transfer. It is interesting to study which of these phases is the bottleneck on the Vivante GPU and the Nvidia Tesla GPU. This is interesting to study because an embedded GPU has a very different system-on-chip based interconnect with the host CPU when compared to high-end GPUs that typically communicate with the CPU via PCI Express bus. Hence, all our experiments also focused on quantifying the overhead of GPU-CPU data transfer. We would like to note that we have, deliberately, not overlapped the data transfer and the GPU execution phases in either the Fermi or the Vivante GPU because of our goal to measure them separately.

F. Measuring the Energy Consumption

To study energy and power consumption, we first measure the average current consumed by the whole system (the board). In fact, in mobile devices where such platforms will be used, it is very desirable to measure the impact of the GPU/CPU implementation on the complete system. It is important to note that peripherals not used in our benchmarks such as displays have been disabled during all our experiments. We took the following two steps to reduce inaccuracies in estimating the current consumed by the GPU and the CPU while running our implementations.

First, we attempt to isolate the current drawn by the benchmarks from the current drawn by other system programs such as operating system routines as well as other hardware components. Towards this, after booting up the board, we measure its stable current without running any application program. This measurement gives us the idle current I_{base} consumption by the board. Thereafter, we execute each benchmark on different devices (GPU, CPU or both) and measure the current $I^{device}_{benchmark+base}$. The difference between $I^{device}_{benchmark+base}$ and I_{base} gives us the "extra" current consumed by the program when implemented on that device. This is the current that we are interested in because it reflects the current consumed by the system due to the implementation of the benchmark on a particular device (i.e., GPU or CPU) and we denote it by $I^{device}_{benchmark}$.

Second, we measure $I^{device}_{benchmark}$ by executing, only the fraction of the algorithm for which we measure execution time, in an infinite while loop. This allows us to measure the average value of the current and eliminates, partially, further inaccuracies. Thus, we measured the *average* current drawn by the GPU and CPU for each program for comparison. For the purpose of this study, where we compare the GPU versus the CPU, energy estimation based on the average current is desirable. This is because the goal of our study is to compare the **relative** energy efficiency between the CPU (sequential, dual-core, multi-core) and the GPU implementations.

It would be certainly interesting to study (i) how optimizations of the GPU code lead to variations in power and energy consumption within a program and (ii) the influence of different components of a program on the current. For this, it would be important to study the current measurement at different instants during the run of the program rather than an average value and is not the focus of this paper.

Calculations: With the measured value of current on GPU, $I^{GPU}_{benchmark}$, the voltage V, and execution time on the GPU denoted by $T^{GPU}_{benchmark}$, we compute the energy consumed as the following :

$$V \times I^{GPU}_{benchmark} \times T^{GPU}_{benchmark} \quad (1)$$

In the case where our application is implemented in a heterogeneous fashion between the GPU and the CPU, $I^{Hetero.CPU}_{benchmark}$ and $T^{Hetero.CPU}_{benchmark}$ refer to the current consumed and the running time required for the component implemented on the CPU.

$$V \times I^{GPU}_{benchmark} \times T^{GPU}_{benchmark} + V \times I^{Hetero.CPU}_{benchmark} \times T^{Hetero.CPU}_{benchmark} \quad (2)$$

The energy consumed on by a CPU-only (e.g., dual-core) is similarly given by:

$$V \times I^{dual-core}_{benchmark} \times T^{dual-core}_{benchmark} \quad (3)$$

We take the ratio of Equation 1 and Equation 3 or the ratio of Equation 2 and Equation 3 depending on our implementation, to quantify the relative energy efficiency between the CPU and the GPU-based implementations. As the voltage V cancels out in the ratio, and since we know the current (see Section V) and the execution times ((see Section IV)) of these terms from measurement, it is straightforward to compute the ratio. The results in the next section are based on the above calculation.

IV. GPU Implementation and Results

In this section, we describe our GPU implementation strategies as well as the experimental results on running times, individually, for each benchmark. Section V provides a more holistic viewpoint of the results.

A. Rijndael Algorithm

1) Implementation: There are several different implementations for the Rijndael algorithm based on different "block cipher modes". Among them, we chose ECB (Electronic Codebook) because it is considered amenable for parallel implementation [9]. Our implementation is based on 256-bit keys with 14 cycles of repetition. Any Rijndael implementation, including ECB, consists of two major phases. The first phase is the key expansion step which is invoked only once at the beginning of the algorithm. It is an inherently serial process and hence, it is performed on the CPU.

The second phase of the Rijndael implementation is the computational bottleneck and hence, it is often chosen as the main kernel for any acceleration. Generally, there are two ways to implement this phase [9]. One is based on lookup tables and the other relies on computations to calculate all the required values. We implemented the Rijndael algorithm based on the lookup tables because we wanted to stress the memory resources of the embedded GPUs, as discussed in our choice of benchmarks in Section III-A.

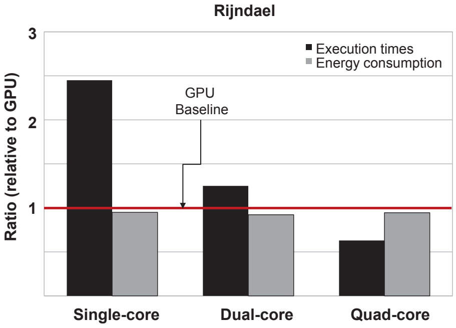
As motivated in Section III-E, to study the impact of local memory on the overall performance, we ran two versions of the lookup table based program on the Vivante GPU as well as the Nvidia Tesla. In the first version with local memory (LM), we utilized the LM to store the lookup table, which would be a conventional GPGPU programming style. The other version directly accessed the global memory for the lookup table.

2) Results: The results of running each program on the two GPUs were quite different and are shown in Figure 1(a). On the Tesla GPU, relinquishing the local memory meant that the execution time of the kernel increased to 10.5 milliseconds (ms) (by almost 7 times) compared to the 1.7 ms in the case when local memory was being utilized. On the contrary, on the Vivante GPU, usage of the local memory led to a performance deterioration by 1.5 times (464 ms with LM compared to 301 ms without LM). This can be explained by the fact that the compiler creates an abstraction for the programmer giving an illusion of "on-chip" local memory (that

Rijndael

	Vivante GC2000		Nvidia Tesla M2050	
Time (ms)	with LM	without LM	with LM	without LM
GPU kernel	**464**	**301**	**1.7**	**10.5**
Data transfer	**78**		**6.5**	
Total (GPU)	542	379	8.2	17
Total (CPU)	915		157	
Speedup	1.7	2.4	19.1	9.2

(a)



(b)

Fig. 1. (a) Execution times (in milliseconds) are listed, both with and without utilizing LM (local memory). **In all tables shown in this paper, for the embedded platform, the speedup over ARM Cortex A9 single-core (respectively, for the Fermi platform, speedup over Intel Xeon single-core) is shown.** (b) Plots showing the relative performance of both execution times and energy. **In all graphs shown in this paper, the red horizontal line at 1 (y-axis) marks the baseline performance by the Vivante GPU compared to the ARM Cortex A9 multi-cores. A graph above this baseline implies lower execution times (or lower energy consumption) by the GPU. For comparison with the CPU, the best GPU-based implementation is chosen.**

does not exist physically). Local memory is, in fact, emulated in the global memory which is the system memory. This means that by moving data from global memory to local memory, the programmer does not move data nearer to the GPU core and instead it only creates a software overhead. As such, programmers used to conventional GPGPU programming need to be aware of such subtle differences with embedded GPUs.

As discussed in Section III-E, it is also very interesting to study the differences between high-end and embedded GPUs with regards to the impact on the data transfer times between the CPU and GPU. Towards this, let us compare the optimized versions from both platforms, i.e., with LM on the Nvidia Tesla GPU and without LM on the Vivante GPU. For Tesla, the ratio of data transfer time to the kernel execution time is 3.82 (6.5 ms/1.7 ms). While for the embedded GPU, the ratio of data transfer time to the kernel execution time is 0.26 (78 ms/301 ms). Thus, for the optimized version on the Tesla GPU, the data transfer phase is the bottleneck while for the Vivante GPU the kernel execution time remains the bottleneck. This is a result of the fact that a system-on-chip embedded GPU like the Vivante GC2000 and its host CPU share the same physical system memory and communicate via a interconnecting bus. On the other hand, the Tesla GPU and its host CPUs have separate DRAMs as main memory and communicate through a bus like the PCI Express.

For all our benchmarks, we also measured the energy consumption by following the method described in Section III-F. Figure 1(b) illustrates the energy consumed by the sequential, dual-core, quad-core implementations (for the ARM cores) as a ratio over the energy consumed by the Vivante GPU implementations. For the Rijndael algorithm, the GPU consumed marginally more energy. Recall that the Rijndael algorithm that we chose to implement was very memory intensive because of extensive use of look-up tables. As global DRAM memory is known to contribute adversely to power consumption in embedded systems, this is not totally unexpected. Figure 1(b) also juxtaposes the relative performance results of both execution times and the energy consumption with respect to the GPU and CPU implementations. This is in order to visualize the tradeoffs between the energy and execution times. Here, it may be observed that in dual-core and single-core settings the GPU presents conflicting tradeoffs as it beats the CPU in performance but performs just poorly with regards to energy.

To compute relative energy consumption, we need execution time and the current drawn (see Section III-F). The execution times are shown in Figure 1(a) while the current drawn is listed in Figure 7.

B. Bitcount

1) Implementation: Given a set of integers in an array as an input, the *bitcount* algorithm gives the total number of set bits in the array as an output. For our GPU implementation, we chose what is known as the Sparse algorithm. For each integer x in the array, in Step I this algorithm counts the number of iterations of the following $while$ loop: $while\ ((x = x\&(x-1))! = 0)$, which gives the number of set bits in x. In Step II, the algorithm computes the sum of such set bits for all the integers in the given array. Both Step I and Step II of this GPU implementation has interesting features as explained below.

There is a limiting factor in Step I which prevents the application from reaching high speedups. This factor is uneven distribution of workload between different threads within a thread group. A Vivante thread group is conceptually similar to “waves” on AMD GPUs and “warps” on NVIDIA GPUs and is essentially the set of threads (work-items in OpenCL terminology) that are run in parallel by the underlying hardware. All threads in the thread group must complete their execution before the GPU computational resources may be allocated to a separate thread group. This is because at a given moment of time, only one thread group can be running on one GPU compute unit.

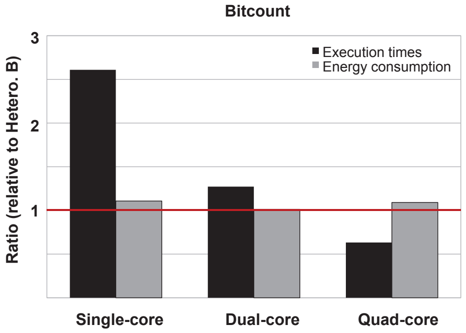
For *bitcount*, threads within a thread group may not complete execution at the same time, thereby leading to underutilization of computational resources. In Step I, the problem arises from the fact that threads which work on integers with relatively less number of set bits, would go through less number of repetitions in the while loop. Thus, they will finish earlier than other threads working on integers with larger number of set bits. However, the threads that complete their job earlier have to stay idle and wait for the other threads in the same thread group to finish.

Recall that Step II of *bitcount* is to sum up the result from each thread. This may be implemented using a common concept in data parallelism known as reduction. In such an implementation, the programmer must enforce synchronization points which means that some threads must idle while other

Bitcount

		Vivante GC2000				Nvidia Tesla M2050		
	Time (ms)	Hetero. A	Hetero. B	with LM	without LM	Hetero.	with LM	without LM
	GPU kernel1 (step 1)	467		467		6.1	6.1	
	GPU kernel2 (step 2)	NA		**260**	**165**	NA	**2.1**	**3.7**
Data Transfer	CPU to GPU	104		104		9	9	
	GPU to CPU	106		ε → 0		5.2	ε → 0	
	CPU (step 2)	**140**	**42**	NA		28.8	NA	
	Total (GPU)	817	719	831	736	49.1	17.2	18.8
	Total (CPU)	1876				552		
	Speedup	2.30	**2.61**	2.26	**2.55**	**11**	**32**	30

(a)



(b)

Fig. 2. (a) Execution times (in milliseconds) for bitcount shown for both, with and without utilizing LM (local memory) as well as two heterogeneous implementations. Speedup over single-core is shown. NA stands for "not applicable". (b) Plots showing the relative performance of both execution times and energy.

threads reach the same synchronization point. This situation can get worse in the case of embedded GPUs where there is no or very little local memory to store intermediate data. To deal with reduction, we propose the use of a heterogeneous implementation by executing the Step II with reduction on the CPU. Towards this, we implemented two versions (i) Heterogeneous A implementation on single-core CPU and (ii) Heterogeneous B implementation on quad-core CPU. For comparison, we also implemented GPU-only versions where the reduction process was performed on the GPU, as in conventional GPGPU programming. We implemented two GPU-only versions — (i) with LM (ii) without LM. In these two methods, reduction was performed by launching a separate GPU kernel (denoted as GPU kernel 2) immediately after the completion of the first kernel which executes Step I. The first method, however, utilized local memory and the second did not.

2) Results: For either GPU-only or heterogeneous implementations, Step I is performed on the GPU and hence, as seen in Figure 2(a), this step takes the same amount of time for all of them.

Let us focus on the results about Step II on the Vivante GPU. In Figure 2(a), the Heterogeneous A and the Heterogeneous B implementations complete Step II in 140 ms and 42 ms respectively on the CPU whereas the implementation with LM on GPU takes 260 ms and implementation without LM takes 165 ms. This clearly shows that, in this case, the *computation* of reduction on the CPU is more efficient compared to an embedded GPU. Yet, executing Step II on the CPU did not actually improve the overall performance for Heterogeneous A because data computed after Step I must be transferred to the CPU memory. This leads to an additional overhead of 106 ms, as shown in Figure 2(a). Hence, Heterogeneous A takes 246 ms for Step II which is slower than the GPU implementation without LM that takes 165 ms.

The Heterogeneous B implementation, however, can compensate for the data transfer latencies by sheer acceleration of the execution time which is only 42 ms by using 4 CPU cores. As such, Heterogeneous B implementation takes 148 ms for Step II which is the best amongst all. Thus, the best overall performance was delivered, neither by the quad-core CPU nor the embedded GPU but, by the Heterogeneous B implementation using both the quad-core CPU and the embedded GPU. This case study illustrates that, compared to high-end GPUs, GPGPU programming on embedded GPUs involves more intricate optimization strategies.

In contrast, offloading reduction from Tesla GPU to the CPU led to a dramatic performance deterioration. With a heterogeneous implementation (on Tesla GPU and single-core Intel Xeon CPU), the speedup over sequential implementation on Xeon CPU was 11×. With GPU-only implementation, the best speedup over the Xeon CPU implementation was 32 times (obtained with LM). In fact, just the data transfer time (5.2 ms) from Tesla GPU to the Xeon CPU in the heterogeneous version took longer than the execution time (2.1 ms) of the reduction on the Tesla GPU.

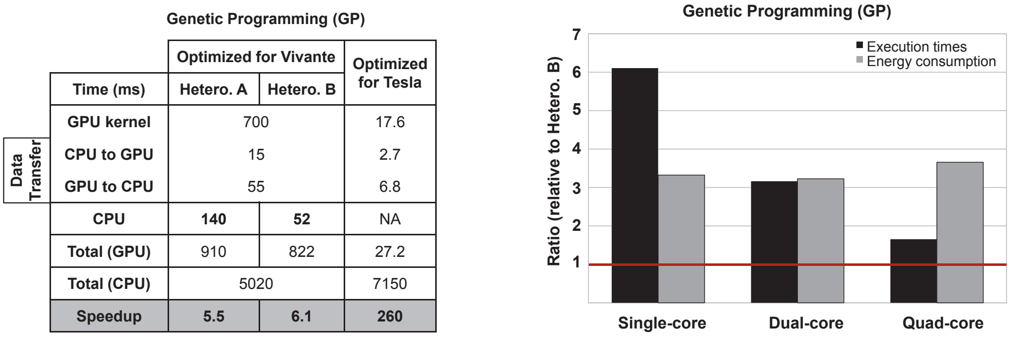
Note that performing reduction without LM on Tesla has exactly an opposite effect compared to the embedded GPU. On the Tesla GPU, without LM, there is 75% performance deterioration (from 2.1 ms to 3.7 ms) compared to the case with LM while the embedded GPU implementation without LM performs about 60% better (from 260 ms to 165 ms).

Finally, to study the impact of synchronization in Step I, we generated a special set of inputs where all the bits were set to 1. In this case, there will be no idling for the threads and thus, there will be no performance loss due to synchronization. The resulting kernel execution time on the Vivante GPU (GPU kernel 1) was faster compared even to the quad-core. This runs counter to the general case where the *bitcount* on GPU is worse than the quad-core and illustrates the bottleneck that arises due to synchronization on GPUs.

Figure 2(b) shows the relative energy and execution times of our Heterogeneous B implementation with respect to various CPU settings. Even if our proposed implementation involves quad-core in Step II, it beats the CPU-only solution in all cases when it comes to energy consumption. This shows the energy efficiency delivered by the GPU in Step I. The other important observation is that, there arises a tradeoff when comparing our Heterogeneous B implementation with quad-core CPU. The quad-core implementation outperforms the GPU in running times but consumes marginally more energy.

C. Genetic Programming

1) Implementation: We have chosen the Intertwined Spiral problem to be solved using genetic programming. Among the step, involved in genetic programming, fitness measurement of individuals in the population is known to be the computational bottleneck and we chose this as the kernel to be accelerated.



Genetic Programming (GP)

		Optimized for Vivante		Optimized for Tesla
	Time (ms)	Hetero. A	Hetero. B	
	GPU kernel	700		17.6
Data Transfer	CPU to GPU	15		2.7
	GPU to CPU	55		6.8
	CPU	**140**	**52**	NA
	Total (GPU)	910	822	27.2
	Total (CPU)	5020		7150
	Speedup	**5.5**	**6.1**	**260**

(a) (b)

Fig. 3. (a) Execution times (in milliseconds) for genetic programming. Speedup over single-core is shown. NA stands for "not applicable". (b) Plots showing the relative improvements of both execution time and energy.

Our model to solve the genetic programming consisted of the following. The program population consisted of 500 individuals. Each individual, based on a mathematical expression that is initially generated randomly, computes whether a given x-y co-ordinate belongs to one of the two spirals. The expression comprises of arithmetic functions of addition, subtraction, multiplication, division as well as two trigonometric functions, sine and cosine. The maximum size of the expression (individual) was allowed to be 200.

It is important to note two important differences between our Nvidia Tesla and our Vivante GPU implementations. For the Tesla GPU, we developed a version of the genetic programming application based on conventional optimization for high-end GPUs, e.g., we used local memory to reduce the memory bandwidth bottleneck as much as possible. However, as already described in Rijndael and *bitcount* implementations, usage of local memory on the Vivante GPU leads to performance deterioration. This holds true for the genetic programming case study as well. Instead of reproducing similar results for local memory here, we want to focus on other issues. Hence, our implementation of genetic programming on the Vivante GPU, that is discussed below, is the one without local memory.

Second, for the Tesla GPU, the entire fitness function, was implemented on GPU. However, on the Vivante GPU it turned out that the size of the kernel was larger than the size of the GPU instruction memory. Hence, we propose a heterogeneous solution by splitting the kernel into two components. The first component runs on the Vivante GPU while the second one runs on the CPU. For the second component on the CPU, we developed two versions — (i) Heterogeneous-A implementation on single-core and (ii) Heterogeneous-B implementation on quad-core. It is interesting to note that the second component involves a code fragment that performs reduction.

2) Results: On the Tesla GPU, our application achieved over 250× speedup compared to an Intel Xeon single-core. On the i.MX6 platform, our proposed implementation Heterogeneous-A (on Vivante GPU and a single core ARM CPU), gave us 5.5× speedup over a sequential single-core ARM CPU implementation (Figure 3(a)). We would like to point out that, on the Fermi, for a similar version heterogeneous implementation (on Tesla GPU and single-core Xeon CPU), the performance acceleration is reduced to 60× (from 250× achieved in the original the Tesla-only optimized version). This example, again, illustrates that different optimizations must be adopted for high-end and embedded GPUs.

In Heterogeneous-B, to further increase the speedup we utilized the potential parallelism from using multiple cores to reduce the execution time of the second component of the fitness function (reduction as well as some other selected subtasks). Thus, in Heterogeneous-B, we have truly distributed the kernel workload across the GPU as well as all the CPU cores. The Heterogeneous-B version on the Vivante GPU was faster than the Heterogeneous-A implementation by 10% (822 ms compared to 910 ms). It delivered 6.1× speedup over a single-core CPU implementation.

Figure 3(b) juxtaposes the relative energy and execution times between our Heterogeneous-B implementation and CPU implementations. It may be noted that Heterogeneous-B obtains 1.65× speedup even over a quad-core implementation and beats the CPU with respect to energy consumption in all the settings.

D. Convolution

1) Implementation: We implemented a standard code for the convolution [3]. As it has been widely discussed in the literature, we only provide a high-level sketch in the following. The code [3] includes four nested *for* loops. The convolution kernel is a natural candidate for GPU parallelism because it works by iterating over each pixel in the source image. The two outer loops iterate over each source pixel in the source image. The filter is applied to the neighboring pixel of each source pixel in the two inner loops. The values of the filter multiply the pixel values that they overlay and then a sum of products is taken to produce a new pixel value.

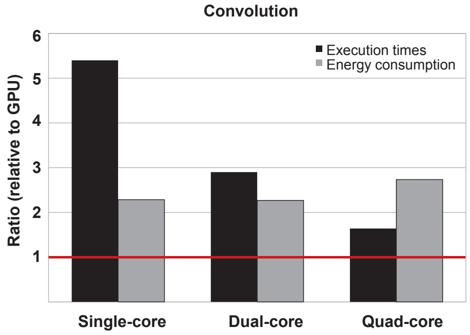
2) Results: The results for convolution are shown in Figure 4(a). These results are in line with expectations for an application with high compute-to-memory access ratio. In fact, as we observe in Figure 4(b), the implementation on the embedded GPU (that is a GPU-only solution) outperforms even the quad-core implementation. With respect to energy consumption, the GPU is better than CPU in all cases.

We note that similar to the results from the Rijndael implementation, *bitcount* and genetic programming, the data-transfer is a relatively larger bottleneck on the high-end Tesla GPU compared to the Vivante GPU. With 13.5 ms + 29 ms, for the Tesla GPU, data transfer contributes to 80% of the overall execution time (53 ms). In contrast, for the Vivante GPU, with 143 ms + 594 ms, data transfers contributes to only 27% of the overall execution time (2703 ms).

Convolution

	Time (ms)	Vivante GC2000	Nvidia Tesla M2050
	GPU kernel	1966	10.2
Data Transfer	CPU to GPU	**143**	**13.5**
	GPU to CPU	**594**	**29**
	Total (GPU)	2703	53
	Total (CPU)	14780	2166
	Speedup	5.45	40

(a)



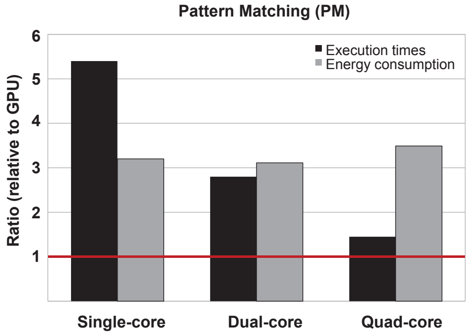
(b)

Fig. 4. (a) Execution times (in milliseconds) for convolution. Speedup over single-core is shown. (b) Plots showing the relative improvements of both execution time and energy.

Pattern Matching (PM)

	Time (ms)	Vivante GC2000	Nvidia Tesla M2050
	GPU kernel1	2.03	0.03
	GPU kernel2	5.59	0.03
	GPU kernel3	1.18	0.02
Data Transfer	CPU to GPU	1.73	0.37
	GPU to CPU	0.41	0.04
	Total (GPU)	10.94	0.49
	Total (CPU)	58.91	3.4
	Speedup	5.38	6.9

(a)



(b)

Fig. 5. (a) Execution times (in milliseconds) for pattern matching. Speedup over single-core is shown. (b) Plots showing the relative improvements of both execution time and energy over GPU.

E. Pattern Matching

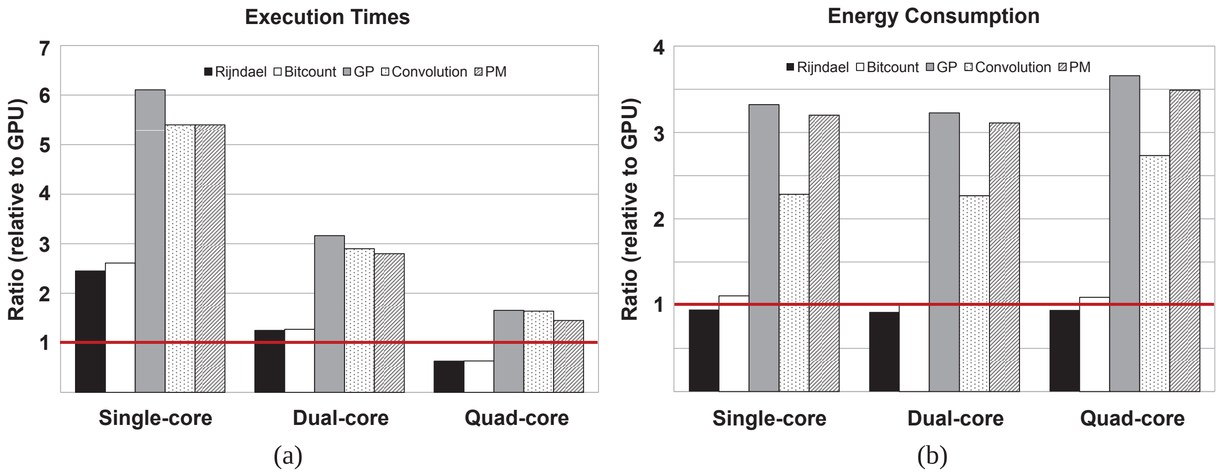
1) Implementation: We selected a pattern matching algorithm from High Performance Embedded Computing Challenge Benchmark from MIT [15]. In essence, the pattern matching algorithm considers two vectors of the same length and computes a metric that quantifies the degree to which the two vectors match.

This benchmark is different from other applications in the sense that there are several distinct sub-components within the computationally heavy part where each component comprises of several nested loops. Implementing this large code into a GPU kernel has several problems. First, the size of the kernel is large and for embedded GPUs, it is important to split it into several fragments such that each of them may fit in the GPU instruction memory. Hence, we split the code into three different kernels which were launched consecutively. Second, given the distinct nature of the different nested loops, each one would perform optimally with a unique number of threads. Third, we did some optimization to save the extra overhead of data transfer between CPU and GPU due to multiple kernels. Our kernels exchanged the required arguments with each other in a way that the outputs of one kernel were stored in one of the arguments and then passed to the subsequent kernel.

2) Results: The results for pattern matching are shown in Figure 5(a). In terms of the speedup obtained on the Vivante GPU, the results are similar to convolution and the GPU outperforms the quad-core Figure 5(b).

We would like to point out that, unlike the previous four benchmarks where the Tesla GPU implementations obtained around an order of magnitude speedup, in this case the Tesla implementation obtained a speedup of around 6.9×. This is explained by the fact that the experiments were performed with the data set available with the High Performance Embedded Computing Challenge Benchmark [15]. The size of this data set was not large enough to keep all the Tesla GPU computational resources busy, leading to under-utilization. However, the data set was large enough for the Vivante GPU cores to be fully engaged and hence, we observe that the results for the Vivante implementations are quite good compared to the other four benchmarks.

Finally, we would like to mention that the first two kernels in this implementation also involve a GPU fragment that needs to perform reduction. However, our experiments showed that offloading this to the CPU (similar to the *bitcount* and the genetic programming implementations) actually led to severe performance deterioration. This is because, unlike *bitcount* and the genetic programming, the output from reduction in each of these kernels are subsequently used by the GPU. If implemented in the CPU, this creates an extra overhead of transferring data from GPU to CPU plus executing it on the CPU plus writing back the results from CPU to GPU. This overhead turned out to be substantially more than having it processed locally on the GPU. Hence, we propose not to have a heterogeneous implementation in the case of pattern matching.



(a) (b)

Fig. 6. Graphs illustrating (a) the execution times of the benchmarks as well as (b) the energy consumed by the benchmarks on the CPUs relative to the GPU implementation.

Current (mA)	$I^{GPU}_{benchmark}$	$I^{Heter.\ quad}_{benchmark}$	$I^{single\text{-}core}_{benchmark}$	$I^{dual\text{-}core}_{benchmark}$	$I^{quad\text{-}core}_{benchmark}$
Bitcount	360	650	160	300	650
Convolution	390	NA	165	305	650
Rijndael	395	NA	155	295	600
GP	270	650	160	300	650
PM	300	NA	160	300	650

Fig. 7. Measured current for all benchmarks in milliamperes.

V. Discussion

This paper attempted to answer the following question: given that resource-constrained embedded GPUs co-exist on the same chip with powerful multi-core platforms, are they suitable for non-graphics workloads? The results, summarized with a holistic view in the next two sections, lead us to conclude that embedded GPUs deliver performance benefits as well as energy efficient solutions. However, our study leads us to the conclusion that embedded GPUs have several constraints and designers must make intelligent tradeoffs to extract the maximum performance benefits. Based on our experience and results, in Section V-C we describe few challenges that arise due to these constraints.

A. Execution Times

Figure 6(a) illustrates the execution times of the single-core, dual-core, quad-core implementations as a ratio over the execution time taken by our GPU/heterogeneous implementation. The results (Figure 6(a)) show that for **all** applications, our implementation ran faster than the single-core and the dual-core implementations. In fact, for convolution, pattern matching and genetic programming, our GPU-based implementation achieves more than (i) 5 times speedup over single-core, (ii) 3 times over dual-core. Recall that convolution, pattern matching and genetic programming are all compute intensive kernels.

Even for the quad-core comparison, our GPU-based implementation is better for three of the applications. Only *bitcount* and Rijndael ran in shorter time in a quad-core setting than our GPU implementation. Also, the severe memory limitations, including a small cache, in the embedded GPU may cause frequent accesses to global memory leading to poor performance for Rijndael. Bitcount has relatively less memory accesses than Rijndael yet, it suffers on GPUs because of delays in hardware synchronization within threads of a thread group. It is important also to note that the Vivante GPU (GC2000) has only one integer pipeline per core, but it has 4 floating point pipelines per core. As such, bitcount and Rijndael, with their integer calculations, cannot benefit from this hardware feature.

Also, we would like to note that for the 5 benchmarks, the acceleration over single-core achieved on the optimized Tesla code was up to couple of orders of magnitude (up to 260×). Not unexpectedly, embedded GPUs may not yet deliver similar speedups as high-end GPUs. Finally, note that as opposed to the Tesla GPU where CPU-GPU data transfer is a big bottleneck, on embedded GPU platforms the bottleneck might be GPU computational power, particularly observed in the Rijndael, pattern matching and convolution.

B. Energy Consumption

Figure 6(b) illustrates that our GPU-based implementations delivered more energy efficient solutions in most cases. GPUs consume less energy inspite of the fact that they draw larger currents (Figure 7). This is because applications on GPUs complete much faster. Recall from Equation 1, that energy is the product of the current and the execution times. Hence, relatively longer running times (due to less speedups) (Figure 6(a)), *bitcount* and Rijndael implementations on the GPU did not show significant energy savings.

It may be observed that with increasing number of cores, the energy consumption does not increase substantially. With increasing number of cores, higher amounts of current are drawn which is shown in Figure 7. Yet, the execution times decrease with increasing number of cores thereby stabilizing the rate of increase of energy consumption. Along this line, note that the GPU implementation typically draws more current (Figure 7) than dual-core or single-core but it is sill more energy efficient.

C. Challenges for GPGPU Programmers

Towards heterogeneous computation: It is not straightforward to take the mapping decision of a given application to a GPU or a multi-core CPU while optimizing performance. Some applications, like Rijndael, are faster on GPUs compared to single-core or dual-core implementations but are slower than quad-core implementation (Figure 6(a)). On the other hand, applications like pattern matching and convolution were better on GPUs even when compared to the quad-core implementation. Furthermore, our case studies with the Rijndael algorithm and *bitcount* showed that there are settings where the GPU and the CPU present conflicting tradeoffs between optimizing energy and execution times. Thus, going forward, heterogeneous platforms with GPUs and CPUs seem an attractive solution for different kinds of applications.

Furthermore, our results on genetic programming and *bitcount* illustrate that for some applications a GPU-only or a multi-core-only solution might be sub-optimal. For instance, in parallelizing “reduction”, high-end GPUs can deliver superior performance compared to CPUs. Yet, in contrast, for certain applications on embedded GPUs, it might be that the “reduction” should be performed on the CPU while the rest of the data parallel code should run on GPU for best performance (as we showed in *bitcount* and genetic programming).

Registers: Registers are at a premium in embedded GPUs, because manufacturers can only put so much registers into the silicon at a given cost. Vivante GC2000, for instance, has a register size of 2KB in each core (with 4 cores in total) in contrast to 128KB registers per multiprocessor in Tesla M2050 (with 14 multiprocessors in total). This imposes serious restrictions on the performance improvements that may be achieved. First, the restrictions on registers have a strong correlation with the number of thread groups. Ideally, each GPU core should host a large number of thread groups. Then, whenever the currently executing thread group is waiting for a memory access to be completed, the scheduler can switch to an alternative thread group that is “active” and ready to run. Limited number of registers can limit the total number of “active” thread groups that in turn limits the ability of the scheduler to hide the memory latencies, thereby adversely impacting the running times.

Second, for kernels that require more registers than available in the hardware, the GPU compiler might instead push a register to be used from GPU memory (global memory), which will also make the program very slow.

Size of the kernel: The size of the kernel matters in embedded GPUs because of limited instruction memory. For instance, the Vivante GC2000 can accommodate up to 512 instructions only and kernels that do not fit can not be supported at all. This will be relaxed in the next generation of embedded GPUs. However, limited size of instruction cache would mean that kernel size might still impact the performance as we move towards implementing larger portions of an application on the GPU in future.

Secondly, larger programs will need to keep more “state”, i.e., more registers to keep the state. As discussed above this has a significant impact on the performance via the number of thread group that may be launched. Finally, larger kernel would typically mean large data sets which can easily lead to poor performance on the limited L1 cache of embedded GPUs. In light of the above challenges, embedded GPU programmers must decide whether it is profitable, from a performance perspective, to split a large kernel into multiple kernels considering that this will also increase kernel invocation overhead. The other alternative would be to split the kernel between the CPU and the GPU. Recall that our implementation of the pattern matching algorithm and genetic programming benefited from these ideas, respectively.

VI. Conclusion

As far as we are aware, this is the first paper to quantitatively evaluate both energy and execution times for a range of non-graphics workloads on an embedded GPU. As architecture for embedded GPUs evolve in future, we imagine a new wave of algorithms that was not possible so far. Consider, for instance, computationally heavy meta-heuristics like evolutionary algorithms, simulated annealing and tabu search. The use of such methods is not popular in online optimization methods because it will take unacceptable amount of time to complete the algorithms. However, this assumes the use of single-core CPUs or at most a handful of cores. The encouraging results we achieved on genetic programming is only a small step in this direction and lot of work must be done before this can be realized in practice.

References

[1] S. Collange, D. Defour, and A. Tisserand. Power consumption of GPUs from a software perspective. In *International Conference on Computational Science*, 2009.

[2] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.

[3] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous Computing with OpenCL*. Morgan Kaufman, 2011.

[4] K. Gupta and J. D. Owens. Compute and memory optimizations for high-quality speech recognition on low-end GPU processors. In *International Conference on High Performance Computing*, 2011.

[5] M. R. Guthaus, J.S. Ringenberg, and D. Ernst. Mibench: A free, commercially representative embedded benchmark suite. In *Workshop on Workload Characterization*, 2001.

[6] S. Huang, S. Xiao, and W. Feng. On the energy efficiency of graphics processing units for scientific computing. In *International Symposium on Parallel&Distributed Processing*, 2009.

[7] J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, 1992.

[8] J.R. Koza, M.A. Keane, M.J. Streeter, W. Mydlowec, J. Yu, and G. Lanza. *Genetic Programming IV : Routine Human-Competitive Machine Intelligence*. Springer, 2003.

[9] D. Le, J. Chang, X. Gou, A. Zhang, and C. Lu. Parallel AES algorithm for fast data encryption on GPU. In *International Conference on Computer Engineering and Technology*, 2010.

[10] J. Leskela, J. Nikula, and M. Salmela. OpenCL embedded profile prototype in mobile device. In *Workshop on Signal Processing Systems*, 2009.

[11] X. Ma, M. Dong, L. Zhong, and Z. Deng. Statistical power consumption analysis and modeling for GPU-based computing. In *Symposium on Operating Systems Principles*, 2009.

[12] Mali Graphics Hardware. /www.arm.com/products/multimedia/mali-graphics-hardware/index.php.

[13] S. Mu, C. Wang, M. Liu, D. Li, M. Zhu, X. Chen, X. Xie, and Y. Deng. Evaluating the potential of graphics processors for high performance embedded computing. In *DATE*, 2011.

[14] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[15] J. Kepner R. Haney, T. Meuse and J. Lebak. The HPEC challenge benchmark suite. In *High-Performance Embedded Computing Workshop*, 2005.

[16] J. W. Sheaffer, K. Skadron, and D. P. Luebke. Studying thermal management for graphics-processor architectures. In *International Symposium on Performance Analysis of Systems and Software*, 2005.

[17] Product Brief: Vivante Graphics Cores. www.vivantecorp.com/Product_Brief.pdf.

[18] ZMS-40 StemCell Processor. www.ziilabs.com/products/processors/zms40.php.