

Pattern Matching in OpenCL: GPU vs CPU Energy Consumption on Two Mobile Chipsets

Elena Aragon¹ Juan M. Jiménez¹ Arian Maghazeh¹
Jim Rasmusson² Unmesh D. Bordoloi¹

¹Department of Computer and Information Science, Linköpings Universitet, Sweden

²Sony Mobile Communications, Sweden

{elena.aragon, juan.jimenez, arian.maghazeh, unmesh.bordoloi}@liu.se
jim.rasmusson@sonymobile.com

ABSTRACT

Adaptations of the Aho-Corasick (AC) algorithm on high performance graphics processors (also called GPUs) have garnered increasing attention in recent years. However, no results have been reported regarding their implementations on mobile GPUs. In this paper, we show that implementing a state-of-the-art Aho-Corasick parallel algorithm on a mobile GPU delivers significant speedups. We study a few implementation optimizations some of which may seem counter-intuitive to standard optimizations for high-end GPUs. More importantly, we focus on measuring the energy consumed by different components of the OpenCL application rather than reporting the aggregate. We show that there are considerable energy savings compared to the CPU implementation of the AC algorithm.

1. INTRODUCTION

Our work is motivated by the arrival of OpenCL-enabled GPUs (sometimes called GPGPUs - General Purpose GPUs) in mobile platforms that now gives us an opportunity to program embedded and mobile devices in the spirit of heterogeneous computation. However, unless the full power of GPU compute on *low-power* platforms can be utilized, the full potential of heterogeneous computation will remain untapped. Despite recent fledgling work in this direction, the question — whether (and what kind of) non-graphics workloads may benefit from mobile GPUs — has largely remained open. This question must be studied while keeping in mind that powerful multi-core CPUs are available on the same chip as the GPUs, and the CPUs are already a promising choice.

Conventional application domains targeted for GPU compute on mobiles include image processing, augmented reality, and computational photography. In this work, however, we study the suitability of GPU compute for an application in the security domain. In this regard, we chose the Aho-Corasick (AC) algorithm that is widely used in intrusion detection. The AC algorithm is a pattern matching algorithm that has been utilized in other domains as well, such as detecting plagiarism, digital forensics, text mining and so on. Given stringent requirement on energy consumption in mobile platforms, studying the impact of our implementation on the energy consumption is a prominent component of this paper.

2. RELATED WORK

Recently, there has been tremendous interest in developing parallel versions for the AC algorithm. The results by

Lin et al. [4] are the most recent ones and they have reported improvements over others [12], [9], [10]. For our case study, we implement the algorithm proposed by Lin et al. [4] but our implementation differs in the following ways. First, our implementation is in OpenCL in contrast to the original CUDA implementation [4]. Second, if their algorithm is implemented without any changes, it will lead to poor performance on mobile GPUs. As such, we perform a series of optimizations that are specific to mobile GPUs to maximize performance benefits. To the best of our knowledge, no results have been reported regarding the implementation of this algorithm on mobile GPUs.

In fact, it is only recently that, applications on mobile and embedded GPUs have triggered interest. In a recent paper, Gupta and Owens [2], discussed strategies for memory optimizations for a speech recognition application targeting a low-end resource constrained GPU. We also note that Mu et al. [6] implemented the benchmarks from High Performance Embedded Computing Challenge Benchmark from MIT [7] on a GPU. However, none of these papers discuss the impact of their algorithms on power or energy consumption. In fact, they evaluated their performance results on GPUs that are not targeted towards low-power devices such as hand-held smart phones.

Finally, we would like to mention that few papers have indeed reported energy consumption on mobile GPUs [1, 8, 11, 5, 3] but they did not focus on the AC pattern matching algorithm. Moreover, unlike them, we study the power consumption at a more fine-grain level rather than only reporting the aggregate values. Taking the AC algorithm as an example, we show the variations in the energy consumed during various phases of an OpenCL application.

3. THE AHO-CORASICK ALGORITHM

We provide a quick overview of the AC algorithm. As an input, the algorithm is given several strings, also called the dictionary and an input text stream. As the output, the algorithm reports all matches in the text stream with the strings in the dictionary.

The algorithm proceeds in the following fashion. First, the AC algorithm combines all the input patterns and generates an automaton. The algorithm, then, traverses the input text character by character, and the finite state machine takes a transition in the automaton corresponding to the input character. Each state in the automaton is also associated

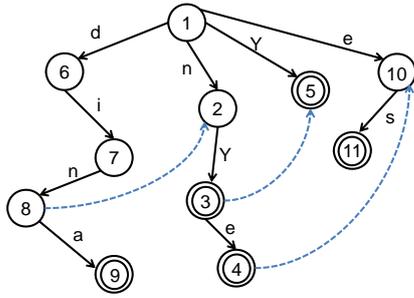


Figure 1: Automaton that accepts “nYe”, “Y”, “dina” and “es”.

with a *failure transition*. Whenever there is no match with the input character, the machine moves to the destination pointed by the failure transition. When it reaches a “final” state, the algorithm finds one matched pattern.

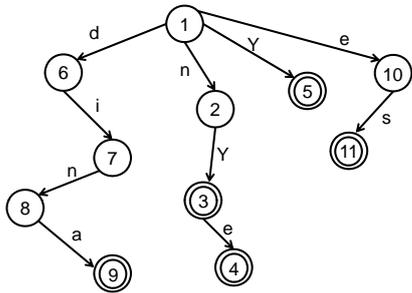


Figure 2: Failureless automaton for the GPU to accept “nYe”, “Y”, “dina” and “es”.

Figure 1 illustrates the automaton built for the patterns “nYe”, “Y”, “dina” and “es”. The circles with double borders are the final states, the solid lines are the transitions labeled with the characters on which they are fired and the dotted lines are the failure transitions. The states 9, 4, 5 and 11 are the final states respectively for the patterns “dina”, “nYe”, “Y”, and “es”. Note that state 3 is also a final state. This is because “Y”, a substring of “nYe” is a valid pattern. Each state without a outgoing failure transition has a default failure transition to the root state 1. To avoid over-crowding, these transitions are not explicitly shown.

Consider the input text as “nYeXs”. To start, the machine will move from state 1 to state 2 and then from state 2 to state 3 with the inputs “n” and “Y”. At state 3, it identifies the match with character “Y”. With the next character “e”, the machine moves to the next final state 4 and identifies a match with the pattern “nYe”. However, with “X” in state 4, the machine takes the failure transition and moves to state 10. Still there is no match and the machine moves back to state 1. There are no further legal moves from the root with “X” or with the last character “s” and the algorithm terminates.

3.1 Parallel Failureless Automaton

In a recent paper, Lin et al. [4] proposed a parallel version of the above algorithm. The main idea is to associate an individual work-item (or thread in CUDA terminology) to each character of the input stream and identify any pattern

that begins on that character. Looking at our example, 5 work-items will be involved for the input string “nYeXs”. The work-items associated with the characters “n” and “Y” will identify the patterns “nYe” and “Y” respectively. However, this parallel algorithm needs a new automaton where failure transitions have been removed. Thus, the automaton shown in Figure 1 will now appear as shown in Figure 2.

Removing the failure transitions is necessary to avoid redundancy. Let us consider a string “dinY”. If we use the traditional automaton from Figure 1, the work-item associated with character “d” will report a match “Y”. The work-item starting at “Y” will also report a match. This duplication is avoided with the new automaton. For more details on the algorithm, we refer the interested reader to the paper [4]. Our OpenCL implementation on the GPU is based on the above algorithm.

4. MOBILE CHIPSET ARCHITECTURES

The two mobile test devices on which we tested our OpenCL implementation are the Sony Xperia Z Ultra and the Arndale development board. These two devices have mobile chipsets from two different vendors, Qualcomm and Samsung. The Sony Xperia Z Ultra is based on the Qualcomm MSM 8974 chipset (also called Snapdragon 800). It has a 2.26 GHz quad-core CPU, the Krait 400, and a quad-core GPU, the Adreno 330 running at 450MHz. We will refer to this board as Xperia in rest of the paper. The Arndale board is based on the Samsung Exynos 5250 that has a 1.7 GHz dual-core CPU, the ARM Cortex-A15, and a quad-core GPU clocked at 533MHz, the ARM Mali T604. We will refer to this board as Arndale. Both chipsets have interesting similarities and interesting differences. The CPUs from the two vendors are both based on the ARMv7 ISA (Instruction Set Architecture). ARMv7 has been around for a couple of years now but the implementations have evolved significantly over the years. The quad-core Krait 400 CPU is Qualcomm’s version of a modern ARMv7 design whereas the dual-core Cortex-A15 is a modern ARMv7 design from ARM themselves.

The GPU architectures are quite different though. Adreno 330 is based on recent compute capable (OpenCL capable) architecture from Qualcomm. Its architecture is somewhat similar to established desktop GPUs from vendors like Nvidia and AMD. These architectures execute a cluster of work-items, so-called *sub-group*, (warps in CUDA terminology) together in lock-step (32 work-items for Adreno), and have some advantages in that they have the potential to be more silicon area- and power-efficient. On the downside, warp based architectures experience performance degradation when running divergent code because different work-items take different execution branches and the total execution time is determined by the sum of the execution times of all branches.

The Mali 604 is a recent compute capable GPU design from ARM. Unlike Adreno 330, it is not based on sub-groups. Instead each thread has its own program counter, more like a CPU, and it has no penalty executing divergent code.

5. OPTIMIZATIONS ON MOBILE GPUS

The original PFAC algorithm was optimized for a high-end GPU. Some of the optimizations that were used are:

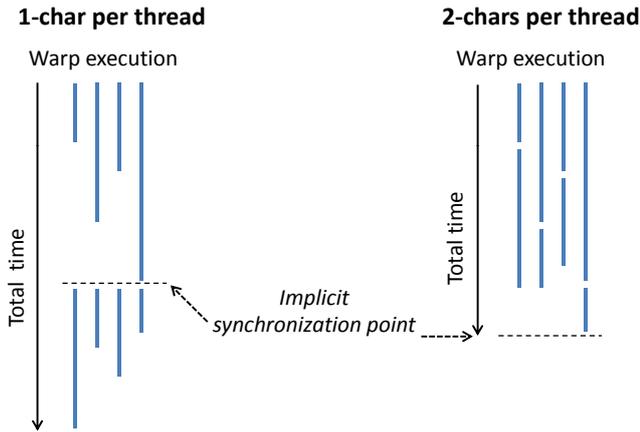


Figure 3: Increasing the workload per work-item helps to reduce the imbalance workload distribution between the sub-groups (warps).

- The input text is selectively and partially loaded from the global memory to the local memory.
- The transition graph is implemented in the form of a look-up table instead of linked-list based graph implementation.
- The first row of the transition table is loaded into the local memory.
- The transition table is stored in texture memory which is a cache optimized memory.

However, using an implementation based on the above optimization leads to a very poor speedup on our mobile GPUs. Given the unique characteristics of our mobile GPUs, we apply the following optimizations.

Reduce/Avoid data traffic between the CPU and GPU: In high-end PC like systems, the CPU and the GPU have (physically) separate main memories and data is transferred between them via the PCI Express bus. This is achieved by the OpenCL function called `clEnqueueWriteBuffer` and `clEnqueueReadBuffer`. However, in mobile GPUs, the CPU and the GPU share a unified physical memory. Using `clEnqueueWriteBuffer` and `clEnqueueReadBuffer` in this case injects an unnecessary overhead to copy the data from one portion of the memory to another portion of the same physical memory. This overhead may be avoided by using `clEnqueueMapBuffer`.

Local memory usage: For mobile GPUs, there is often no dedicated on-chip local memory. In those cases, any OpenCL access to the local memory is emulated in the global memory. This often leads to performance deteriorations instead of performance improvement. It may be noted that Adreno 330 actually has 8KB local memory physically on-chip and we carefully customized our code to exploit this minimal amount of local memory. But in our application, it did not bring any benefits, as we will show in Section 7.2.

Work-group size: We also experimented with different work-group sizes for each mobile GPU.

Workload granularity: The original PFAC implementation launches one work-item to process one character of the input text. This work-item follows the pattern in the transition graph that matches with the string starting with this character. As a sub-group or warp (typically 32 work-items) is executed in lock-step, the execution time for the sub-group will be determined by the slowest work-item. As such, different sub-groups in the same work-group are highly likely to consume different amount of execution times (Figure 3).

This imbalance between different sub-groups may be decreased, at least on average, if each work-item is processing more than one character. This increases the total set of characters processed by each sub-group and, given the randomness of the typical text input, increases the chances that the slowest work-item from two different sub-groups take similar execution times and hence less resources are wasted. Figure 3 illustrates a hypothetical scenario with two characters workload per work-item reducing imbalance between the work-items. We experiment with the number of characters that is processed by each work-item to find the one leading to optimized results. Interestingly, as we will see in the results section, this optimization technique will bring different results for the two GPU architectures.

6. METHODOLOGY

The main challenge in computing energy consumption is to measure the current drawn as accurately as possible. Towards this, we utilize the following setup.

6.1 Experimental Setup

Our setup for energy measurement is shown in Figure 4. To measure the current, the input voltage passes through a precision resistor of 10 milliOhms that is connected in series with the device — the Xperia phone or the Arndale board. Any variation in the current drawn by the device will be reflected in the voltage drop across the precision resistor. Note that the amount of the voltage drop is very small and hence, we amplify the voltage 100 times with an amplifier before sending it to the oscilloscope. We then connect the oscilloscope (PicoScope 5243A) to a computer terminal via a USB connection so that all data may be viewed, recorded and manipulated by the PicoScope 6 software.

To perform our measurements for the Xperia phone, we remove the battery from the phone and instead plug in a special adapter board called the dummy battery to the battery connector of the phone. This adapter board has the proper circuitry to allow for an external regulated power supply. As shown in the figure, an external DC power source supplies the voltage (5 Volts) required by the target device.

Given the above, we know that the oscilloscope displays the amplified voltage V_a . Given the resistance value (10 milliOhms) and the voltage magnification factor (100 times), we use the following equation to compute the current drawn by the device:

$$I_{device} = \frac{(V_a/100) \text{ Volts}}{10 \text{ milliOhms}} = V_a \text{ milliAmperes} \quad (1)$$

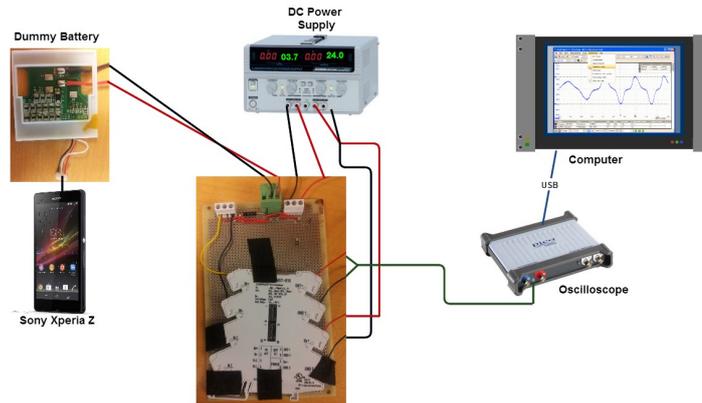


Figure 4: The experimental setup.

We know that energy is given by the product of the current drawn by the application, voltage supply and the execution time of the application. The current drawn is measured as above. The voltage is known because we supply a regulated voltage of 5 Volts while the execution time can be logged in the application itself. It should be noted that we are interested in comparing the relative energy consumptions between the GPU implementation and the sequential CPU implementation.

6.2 Measuring the energy

Our goal is to measure the energy consumed by the OpenCL code of the AC pattern matching application on the GPU and compare it to the energy consumed by the corresponding C code on the CPU. Moreover, we aim to perform a more fine-grained study rather than report the aggregate energy.

Whenever the application executes, there is a spike in the current drawn and this is reflected in the higher voltage levels displayed on the oscilloscope. As the code executes, the oscilloscope records and displays the voltage V_a on the y-dimension as a function of time on the x-dimension. The main challenge is to establish, as accurately as possible, the connection between these spikes and the source of these spikes in the execution of the code. As such, as a first step, we let the device be idle without invoking our code or any other application. This shows the “base” energy consumption by the phone. Hence, when we proceed to measure the energy consumed by the application, we subtract this base energy consumption.

To isolate the energy consumed by different phases of the OpenCL application, we put the device to sleep before and after every phase that we want to measure. For our study, we divide the OpenCL application into four phases. The first phase is related to the administrative part of finding the device, setting up the context and compiling the kernel. The second phase is regarding the transfer of data from the CPU to the GPU, the third phase is related to the execution of the kernel itself and the final phase is writing the data back from the GPU. Finally, we also measure the energy consumption on the CPU.

If the current drawn by these phases is sufficiently high relative to the “base” current, it becomes feasible to measure it

more accurately. To assist us in this, we insert sleep modes between every phase. The sleep modes ensure that the phone is back in the base phase and makes it feasible to identify the current drawn by the phase. To be more accurate, for each phase, we first measure the execution time as logged by the application. This time window is then superimposed with the results plotted on the Oscilloscope. For each OpenCL phase, we validate the execution time reported by the application and the width of the corresponding spike. This helps us to validate that we have correctly identified each of the phase on the plot. Thereafter, we export the data to Matlab where we identify the points where the spikes corresponding to each phase start and end. The area under this curve is computed (giving us the product of current and time) and when multiplied with the supply voltage, it yields the energy consumed by the device for this phase of the OpenCL application. Recall that we subtract the base current drawn by the device during the same window.

We used the test patterns from Snort V2.8 as input benchmark [4]. We generated 1000 test patterns with a maximum size of 128 characters and input text of size 10MB. The finite state machine that was generated contained 27,570 nodes and the GPU required 44MB of memory.

7. RESULTS

This section reports the results obtained by carrying out a wide range of experiments with our implementations on both platforms — Arndale and Xperia. The first set of results (Section 7.1) are related to the plots that are obtained from the oscilloscope and the inferences that may be drawn from them. In Section 7.2, we discuss the second set of experiments that highlights the impact of optimizations that we pursued for the mobile GPUs on execution times and energy consumption. We also report results that compare the mobile GPU with multi-core implementations on the same platform (Section 7.3). Finally, we include a few comments on the CPU cores on both platforms in Section 7.4.

7.1 Energy

Figure 5 shows a snapshot from the oscilloscope for our experiment for two runs of the application on the Arndale board — before and after the optimizations that we discussed in Section 5. The x-axis denotes time and the y-axis denotes the voltage drop across the resistor. As shown in

ARNDALE BOARD							time units are in milliseconds			
OPTIMIZATIONS				RESULTS						
DATA_TX	USE_LOCAL	WG_SIZE	THR_GRAN	WRDEV	KERNEL_EXE	RDDEV	TX_OVH	GPU_TOT	SPEED UP	ENG IMPROV.
no map	yes	128	1	91	295	60	34%	446	4,7	3,3
map	yes	128	1	91	295	6	25%	392	5,4	3,6
map	yes	128	1	91	295	6	25%	392	5,4	3,6
map	no	128	1	91	150	6	39%	247	8,5	8,2
map	no	64	1	91	155	6	38%	252	8,3	8,2
map	no	128	1	91	150	6	39%	247	8,5	8,2
map	no	256	1	91	143	6	40%	240	8,8	8,3
map	no	256	4	91	114	6	46%	211	10,0	9,0
map	no	256	8	91	104	6	48%	201	10,4	9,3
map	no	256	12	91	101	6	49%	198	10,6	9,3
map	no	256	16	91	97	6	50%	194	10,8	9,5

Figure 7: The impact of optimizations on the running times and the energy consumed by the Mali T-604 (Arndale).

SONY XPERIA Z ULTRA							time units are in milliseconds			
OPTIMIZATIONS				RESULTS						
DATA_TX	USE_LOCAL	WG_SIZE	THR_GRAN	WRDEV	KERNEL_EXE	RDDEV	TX_OVH	GPU_TOT	SPEED UP	ENG IMPROV.
no map	yes	128	1	113	208	171	58%	492	2,1	2,7
map	yes	128	1	34	208	0	14%	242	4,3	7,2
map	yes	128	1	34	208	0	14%	242	4,3	7,2
map	no	128	1	34	150	0	18%	184	5,7	10,7
map	no	64	1	34	168	0	17%	202	5,2	10,0
map	no	128	1	34	150	0	18%	184	5,7	10,7
map	no	256	1	34	140	0	20%	174	6,0	11,3
map	no	256	4	34	99	0	26%	133	7,9	13,3
map	no	256	8	34	80	0	30%	114	9,2	15,1
map	no	256	12	34	202	0	14%	236	4,4	10,6
map	no	256	16	34	198	0	15%	232	4,5	10,7

Figure 8: The impact of optimizations on the running times and the energy consumed by the Adreno 330 (Xperia).

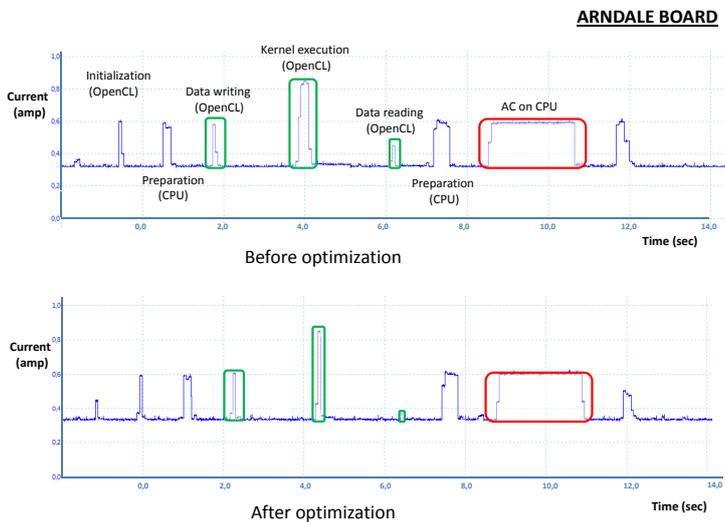


Figure 5: Results showing the energy consumed by different application phases on the Arndale board. The results are shown for OpenCL implementations before and after our GPU optimizations discussed in Section 5.

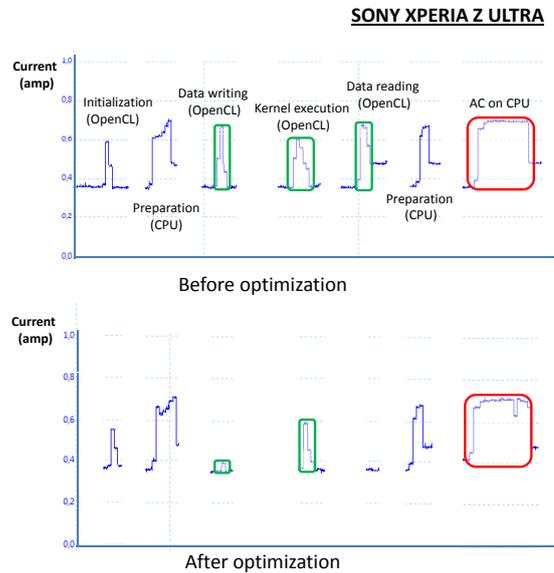


Figure 6: Results showing the energy consumed by different application phases on the Xperia Z Ultra. The results are shown for OpenCL implementations before and after our GPU optimizations discussed in Section 5.

Equation 1, the voltage translates into the current drawn by the device. Recall that the area under each spike represents the energy consumed by the corresponding application phases responsible for that spike.

The first spike in the graph, labeled as “Initialization”, is associated with the OpenCL context creation and compilation of the OpenCL application. The next spike is related to reading the input text and patterns, as well as generating the automaton. These functions are performed on the CPU and are called before launching the GPU. In the figure, this stage is labeled as “Preparation”. Both of these stages, “Initialization” and “Preparation”, need to be performed only

once and hence, the related energy cost must be paid only once. Thereafter, the three spikes highlighted in green show the energy consumed during data transfer and kernel execution on the GPU. Once the GPU run is over, another spike may be observed that is related to “Preparation” on the CPU prior to the sequential invocation of the algorithm. The area under curves marked by green borders and red borders refer to the energy consumed by the GPU and the CPU respectively.

Similarly, Figure 6 shows a snapshot from the oscilloscope

for two runs of the application — before and after the optimizations — on the Xperia mobile platform. The x-axis denotes time but it has been broken down to accommodate the long 'sleep' intervals that were inserted to isolate the different application phases. Unlike the Arndale board, this was necessary here because the current spikes do not fall back to base levels. As such, relatively longer sleep intervals allow the current to settle down to base levels after which we may proceed to the next stage.

The following inferences may be made from these results. First, the overall energy consumed decreases significantly when we compare the optimized and the non-optimized versions. The use of *map*, i.e., the OpenCL function `clEnqueueMapBuffer`, leads to almost negligible energy consumption on Xperia during both writing and reading phases while on Arndale, this is more pronounced only during the reading phase. Actually, it turns out that Arndale also benefits significantly from *map* during writing if the data size is small. Unfortunately, this does not scale to larger inputs. As our experiments are carried out on a relatively large benchmark, our application does not benefit from this optimization during the write phase on Arndale board.

Second, it may be noticed that for both platforms, the peak current consumption during the GPU kernel execution remains the same for both optimized and non-optimized versions of the code.

Finally, it may be also noted that the CPU execution consumes significantly more energy than the GPU. This is mostly the result of the fact that the CPU takes longer time to complete. On Xperia platforms, the peak current drawn by the CPU is higher than the GPU and this may also contribute to the higher energy consumptions. On the Arndale, however, the CPU consumes more energy despite the fact that the peak current drawn by the CPU is less than the one drawn by the GPU.

7.2 Impact of optimizations

Detailed results obtained from the various optimizations that we applied are reported in Figure 7 and Figure 8. Each row shows the impact of the four optimizations (Section 5) on the running times and the energy consumption. The column `DATA_TX` shows whether memory copy is avoided or not. The column `USE_LOCAL` shows whether the local memory was used or not. `WG_SIZE` refers to the work-group size. `THR_GRAN` refers to the work-item (thread level) granularity that was applied.

The results in the next three columns show the breakdown in the times consumed by writing to GPU memory (`WRDEV`), the kernel execution time (`KERNEL_EXE`) and the reading back from GPU (`RDDEV`). `TX_OH` shows the overhead due to data transfer times (as a percentage of the total time) and `GPU_TOT` shows the total time.

The last two columns show the relative speedup with respect to the running times and the relative improvement with respect to the energy consumption when compared to the sequential CPU implementation.

Reduce/Avoid data traffic between the CPU and GPU: Our first optimization is related to the use of *map* (`clEnqueueMapBuffer`) to avoid data transfer. Looking at

the results from the Arndale platform in Section 5, the speedup goes up to $5.4\times$ from $4.7\times$ and energy improvement reaches $3.6\times$ from $3.3\times$. Even higher improvements are noticed in Xperia as the speedup goes up to $4.3\times$ and energy improvement reaches $7.2\times$. The reason is that, on the Arndale board, *map* operation reduced the overhead only in the reading phase (see `RDDEV` column) and did not reduce the overhead in writing phase (see `WRDEV` column) On the other hand, on Xperia, *map* helped reduce the overhead on both phases. This is also reflected in the `TX_OH` column. On Arndale, the transfer overhead reduces from 34% to 25% while on Xperia this reduction is much more dramatic — 58% to 14%.

Local memory usage: Next, as we re-write the code to avoid usage of local memory the speedup goes up to $8.5\times$ and $5.7\times$ respectively on the Arndale and Xperia platforms. Corresponding energy improvements reach $8.2\times$ and $10.7\times$. For the Adreno GPU this result is a bit strange as it actually has dedicated physical local RAMs implemented on chip. Normally on desktop GPUs, using the local RAMs, is a common optimization technique that often brings quite significant speed-ups. For the Mali GPU, this result was more expected as it does not have a local RAM physically implemented on the chip.

Work-group size: The next optimization is related to work-group size and the best results are obtained with a work-group size of 256 on both platforms.

Workload granularity: Finally, we experiment with the workload granularity and as the results show the best results are obtained with a granularity of 16 and 8, respectively on the Arndale and Xperia platforms. A further increase in workload granularity results in saturated performance on Arndale, while on Xperia, a number greater or smaller than 8 yields poorer performance. Interestingly this optimization technique works well for the Adreno GPU which was inline with our expectations. However, above 8 characters per work-item, the execution slows down significantly. Most likely, this is due to the fact that we hit some kind of performance cliff, maybe we run out of registers, and the runtime will have to spill the content of the registers to global memory, and this causes overhead which slows down the execution.

To summarize, on Xperia the most optimized version gives us a speedup of $9.2\times$ and energy savings of $15.1\times$ over sequential implementation. The respective numbers on Arndale are $10.8\times$ and $9.5\times$.

It should be noted that, in the above discussion, other combinations of the work-group size and the thread granularity are possible and we, in fact, explored them. As they turned out to be non-optimal, for the sake of clarity, we have restricted the discussion to the particular work-group size that led to the best results.

7.3 Comparison with multi-core

As Arndale also has a dual-core CPU, we proceeded to compare the Mali GPU with a multi-core implementation. Similarly, Xperia also has a quad-core CPU, and we compared the Adreno GPU with a multi-core implementation. Towards this, we implemented an OpenMP version of our PFAC code. Note that when our OpenMP code runs on a single

ARNDALE	PFAC OPENMP		MOST OPTIMIZED on GPU
	1 CORE	2 CORE	
TIME (ms)	680	620	GPU_KERNEL TIME = 97 (ms) GPU_OVERALL TIME = 194 (ms)
KERNEL SPEED UP	7,0	6,4	
OVERALL SPEED UP	3,5	3,1	
ENERGY IMPROV.	3,3	4,9	

Figure 9: Comparison of the PFAC implementation on Mali GPU with OpenMP implementation on the dual-core CPU.

SONY Z ULTRA	PFAC OPENMP				MOST OPTIMIZED on GPU
	1 CORE	2 CORE	3 CORE	4 CORE	
TIME (ms)	345	175	118	89	GPU_KERNEL TIME = 80 (ms) GPU_OVERALL TIME = 114 (ms)
KERNEL SPEED UP	4,4	2,2	1,5	1,1	
OVERALL SPEED UP	3,0	1,5	1,0	0,8	
ENERGY IMPROV.	5	4	4	4	

Figure 10: Comparison of the PFAC implementation on Adreno GPU with OpenMP implementation on the quad-core CPU.

core, this is a different version than the sequential implementation that we had used so far.

The results are shown in Figure 9 and Figure 10. The GPU provides better speedups and energy savings compared to the dual-core CPU on the Arndale platform. On the Xperia platform, the GPU beats the single-core and dual-core implementations. The GPU is slightly slower than the quad-core implementation. Despite this, from an energy perspective, the GPU is still better than the quad-core CPU by a factor of 4 times.

7.4 Comparing the ARM CPUs

As discussed in Section 4, both two mobile chipsets have CPUs based on ARMv7 ISA. Given that (i) they have different specifications, e.g., clock frequencies, and (ii) the fact Xperia ARM cores have been modified and adapted by Qualcomm, we believe it is not fair to compare them. However, for the interested reader and the sake of completeness, we would like to provide the results of our OpenMP implementations on both platforms. On Xperia, the single-core, the dual-core and the quad-core implementations consumed 0.64, 0.49 and 0.48 Joules, respectively, while on Arndale the single-core and the dual-core implementations consumed 0.94 and 1.41 Joules. It may be observed that the current levels during the CPU execution in Figure 6 and Figure 5, it may be observed that the silicon process node used for the Qualcomm MSM 8974 chipset (TSMC 28nm HPM) probably has higher power consumption than that of the Samsung Exynos 5250 (Samsung’s own 32nm HKMG). In spite of this, the overall energy consumption on Xperia is less than Arndale because the Qualcomm ARM cores on Xperia execute faster. The execution times on Xperia for the single-core, the dual-core and the quad-core are 345, 175, and 89 milliseconds, respectively, while on Arndale the single-core and the dual-core took 680 and 610 milliseconds.

8. DISCUSSION

In this work, we selected the AC algorithm and showed that it also benefits from GPU computing in mobile devices. Our work shows that mobile GPUs are not only suitable for achieving accelerations in running times but they are a promising alternative to save energy. In particular, in spite of the fact that a multi-core CPU implementation might slightly outperform the GPU in terms of speedups,

the GPU implementation may still deliver far improved energy efficiency.

This work may be extended in several directions. First, we need to investigate if further optimizations can improve performance. Second, it will be worthwhile to investigate optimization techniques that minimize power and temperature as well, and not just the overall energy consumption. We are also interested in developing scheduling techniques that will automatically leverage the heterogeneous computational resources while optimizing power or temperature.

9. REFERENCES

- [1] K.T. Cheng and Y.C. Wang. Using mobile GPU for general-purpose computing - a case study of face recognition on smartphones. In *International Symposium on VLSI Design, Automation and Test*, 2013.
- [2] 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.
- [3] M. Huang and C. Lai. Accelerating applications using GPUs on embedded systems and mobile devices. In *International Conference on Embedded and Ubiquitous Computing*, 2013.
- [4] C.H. Lin, C.H. Liu, L. S. Chien, and S.C. Chang. Accelerating pattern matching using a novel parallel algorithm on GPUs. *Transactions on Computers*, 62(10):1906–1916, Oct 2013.
- [5] A. Maghazeh, U.D. Bordoloi, P. Eles, and Zebo Peng. General purpose computing on low-power embedded GPUs: Has it come of age? In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2013.
- [6] 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 *Design Automation and Test in Europe*, 2011.
- [7] J. Kepner R. Haney, T. Meuse and J. Lebak. The HPEC challenge benchmark suite. In *High-Performance Embedded Computing Workshop*, 2005.
- [8] B. Rister, Guohui Wang, M. Wu, and J.R. Cavallaro. A fast and efficient sift detector using the mobile GPU. In *International Conference on Acoustics, Speech and Signal Processing*, 2013.
- [9] A. Tumeo, S. Secchi, and O. Villa. Experiences with string matching on the Fermi architecture. In *International Conference on Architecture of Computing Systems*, 2011.
- [10] G. Vasiliadis and S. Ioannidis. Recent advances in intrusion detection. Lecture Notes in Computer Science, pages 79–96. Springer Berlin Heidelberg, 2010.
- [11] G. Wang, Y. Xiong, J. Yun, and J.R. Cavallaro. Accelerating computer vision algorithms using OpenCL framework on the mobile GPU - a case study. In *International Conference on Acoustics, Speech and Signal Processing*, 2013.
- [12] X. Zha and S. Sahni. GPU-to-GPU and host-to-host multipattern string matching on a GPU. *Transactions on Computers*, 62(6):1156–1169, June 2013.