Performance Characterization of Mobile GP-GPUs

Fitsum Assamnew Andargie
School of Electrical and Computer Engineering
Addis Ababa University
Addis Ababa, Ethiopia
fitsum.assamnew@aait.edu.et

Jonathan Rose
The Edward Roger Sr. Department of Electrical and Computer Engineering
University of Toronto
Toronto, Canada
Jonathan.Rose@ece.utoronto.ca

***Abstract*— As smartphones and tablets have become more sophisticated, they now include General Purpose Graphics Processing Units (GP GPUs) that can be used for computation beyond driving the high-resolution screens. To use them effectively, the programmer needs to have a clear sense of their microarchitecture, which in some cases is hidden by the manufacturer. In this paper we unearth key microarchitectural parameters of the Qualcomm Adreno 320 and 420 GP GPUs, present in one of the key SoCs in the industry, the Snapdragon series of chips.**

Keywords—smartphones; GP GPU; microarchitecture; Adreno GPU;OpenCL

I. INTRODUCTION

Smartphones have progressed dramatically in the last few years, fueled by the exponential advances in semiconductor technology. These advances have enabled a huge number of applications to be developed in a broad range of areas including health, education, sports, music and many more. Some of these applications demand more performance than available in the phone; for example, real-time computer vision applications are often not possible because they require processing of many pixels at acceptable frame rates. These problems may be well addressed by the new General Purpose Graphics Processing Units (GP GPUs) that have become available in the the latest phone core applications processor chip, such as the Nvidia Tegra, Qualcomm Adreno, and the Imagination Technologies PowerVR IP core. GP GPUs have already revolutionized scientific and high performance computing in desktops and servers enabled through the innovation of new architectures [1] and languages such as CUDA [2] and OpenCL [3].

The architecture of all GP GPUs (both desktop, server and the newer mobile GP GPUs) are typically highly data parallel, performing the same computation over different data at the same time. They also make use of many independent threads of execution. In this basic paradigm, the microarchitecture has evolved significantly in the past five years, to include various levels of caching and special memories. The manufacturers of mobile GP GPUs, however, don't describe many of these details nor do they describe or specify the performance of individual communication channels within the processor. These kinds of details, however, are essential for the programmer to optimize the programs written for these devices, where microarchitectural issues such as cache size and memory bandwidth will strongly influence the structure of efficient algorithms [4]. The microarchitectural parameters of desktop/server GP GPUs are well understood and revealed by the vendors [4], whereas mobile GP GPU microarchitectures are far less well documented. The purpose of this paper is to measure key aspects of the microarchitecture and micro-communication channels for the Qualcomm Adreno 320 [5] and 420 GPUs, which exist in the widely used Snapdragon series of SoCs [6] used in many tablets and phones. Understanding these GP GPUs will enable high performance applications to be developed for platforms that harbor them.

II. BACKGROUND

Recent smartphones are equipped with many kinds of compute modalities that can be used to enhance application performance. In the Android platform, most applications are developed using Java that runs on top of Android's Java Virtual Machine called Dalvik [7]. Developing in Java is sufficient for applications that are not compute intensive, but subsequent versions of the Android operating system include a just in time compiler (JIT) that improves runtime performance. For applications that require more performance, compute-intensive portions of programs can be written in C or C++ and integrated through Android's Java Native Interface. In addition, the multiple CPU cores in modern smartphone Application processor chips can be employed through multi-threading; finally there is also a small-scale vector processing capability present in the CPUs that can be utilized for boosting performance -- for example ARM's NEON vector instruction set [8].

The next level of performance improvement can be had through the recent introduction of general-purpose graphics processors (as opposed to dedicated graphics engines) in the mobile processor SoCs. Applications can partition their tasks between the CPU and the GP GPU based on the nature of the task. Serial and task parallel components can be scheduled on the CPU while data parallel parts can run on the GP GPU. In early days of the mobile GP GPUs, one had to reinterpret general problems as graphics problems using graphics libraries, such as Open GL ES. But recently, general propose programming languages for GP GPU development such as CUDA [2] and OpenCL [3] can be utilized.

III. RELATED WORK

Several researchers have previously tried to utilize the mobile GPU to boost the energy efficiency of mobile devices

978-1-4799-7498-6/15/$31.00 ©2015 IEEE

while performing specific applications. Pulli et. al. [9] showed how the original mobile (non-general purpose) GPUs can be used to accelerate a number of vision applications including object detection, object tracking, and scene modeling and augmented reality. They also explored the computational photography applications high dynamic range (HDR) imaging and panorama capture.

Cheng et. al [10] used OpenGL ES to map the face recognition problem to a graphics-rendering paradigm. They implemented the Gabor wavelet computation using the Fast Fourier Transform method, as part of a face recognition computation. The platform used in their investigation was an NVidia Tegra SoC with the CPU running at 1 GHz and the GPU running at 333MHz. The face recognition took about 8.5 seconds on the CPU while taking only 4.6 seconds on the GP GPU while consuming 16.3J of energy in contrast to the CPU only implementation's 29.8J. This shows an almost 2x speed up was gained using the GPU while also lowering the energy consumption by 45.3%.

In similar work done by Rister et. al. [11], the Scale-Invariant Feature Transform (SIFT) detector (which is often used in object detection [12]) was implemented on a mobile GP GPU. In their approach, data was partitioned between the CPU and GP GPU giving the data parallel portion of the work to the GP GPU. In this work, the movement of data between main memory and GPU memory, is carefully minimized, as it is often slow. For example, they reduce the data transfer burden by compressing the image through by pixel reordering. Since SIFT detector works on gray scale images, 4 pixels can be copied into one texture pixel which expects red, green, blue and alpha values. This reduced the data transfer requirement by up to four times. They also report a significant energy consumption reduction (87%) as compared to CPU only implementation when using the CPU+GPU combination.

The SIFT detector was also implemented by Wang et. al. [13] using C++ and OpenCL. In this work, the optimizations mentioned in the work by Rister et. al. were used with the addition of a fast Gaussian blur pyramid generation. With these optimizations, they were able to achieve about 8.5 frames per second for key point detection and 19 frames per second for descriptor generation. A performance speed up of 1.7 times for key point detection as compared to an optimized C++ reference implementation on the CPU was achieved. In addition, energy consumption was reduced by 41%.

Wang et. al. [14] implemented object removal from images using an exemplar-based in-painting algorithm on a mobile GP GPU. The object removal algorithm was implemented as a heterogeneous CPU-GPU application using OpenCL after profiling revealed the bottleneck part of the algorithm. Further optimizations of their implementation used processing of data in vector form and using data sharing by copying to local memory of the GPU. The heterogeneous implementation reduced the runtime required to about 2 seconds on the GPU vs. 398 seconds on the CPU (which was also coded in OpenCL).

In the early days of mobile GPU acceleration the graphics-processing language OpenGL ES was employed to make use of the capabilities of the mobile GPUs. This is difficult, because it requires reinterpreting general problems as graphics problems. The introduction of OpenCL recently has made this part of the problem easier. In addition, Computer Vision problems seem to be the main focus of the literature with regards to utilizing the general-purpose nature of the mobile GP GPUs. The reason behind this trend is that computer vision problems are computationally intensive and are data parallel, for which the GP GPUs are very suitable.

IV. Methodology and Experiments

Recall that the purpose of this work is to measure a specific mobile GP GPU architecture and to understand the capabilities of the underlying hardware so that applications developed for it can be optimized.

We have chosen to measure two generations of Mobile computer systems from Qualcomm (as they are one of the world's largest smartphone chipset vendors): the Snapdragon S4 Pro (APQ8064) first available in 2012 [5] and the Snapdragon 805 (APQ8084) first available in 2014 [6]. We access these through tablet-based mobile development platforms from BSquare and Intrynsic [15]. The S4 Pro applications processor has four processors similar to the ARMv7 architecture which Qualcomm calls the Krait 200 processor. It runs at frequencies up to 1.5 GHz, and contains 2GB of LPDDR2 memory running at 533MHz. The GPU in the S4 Pro is one of Qualcomm's own design, and is called the Adreno 320 GPU and it runs at a clock frequency 325 MHz. The origins of the architecture of this GPU are from the ATI Imageon graphics engines, which Qualcomm acquired in the late 2000's. Figure 1 shows an abstracted view of the processors in Qualcomm's Snapdragon S4 Pro.

The Snapdragon 805 system has four newer-generation Krait 450 processors running up to 2.5 GHz, with 3GB of LPDDR3 memory running at 800MHz. It has a next-generation Adreno 420 GPU running at twice the frequency - 600MHz. In the following experiments we will be measuring the GPUs of both systems, and the contrast and improvements from the first to the second will be illustrated.

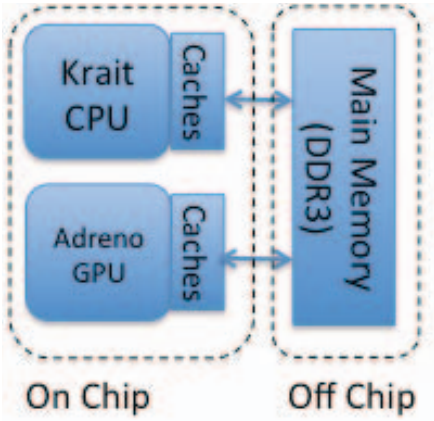

Figure 1 - Abstracted Architecture of Qualcomm Snapdragon SoC

The key parameters that will help a programmer understand the capability of this kind of system, that we will measure, are:

1. The Data transfer rate between Host CPU and GP GPU.
2. The structure of the memory Caching system – how many levels, and size of the cache at each level.
3. The arithmetic computation performance of the GPUs.

Each of the following subsections describes the exact methodology used to measure the above quantities, and then provides the results measured.

A. Data Transfer Throughput Measurement

Historically, the desktop/server GP GPUs had physically separate DDR DRAM memory, which was connected to the physically separate chips that contained the CPU (typically referred to as the host) and the GP GPU. In that context data must be copied to the GP GPU's memory from main memory and returned to the CPU main memory after computations are complete. These data transfers often take up a significant fraction of the computation time. In this section we measure the transfer times for copying between the two memories.

The OpenCL APIs provided for data transfer were used to measure the cpu-to-gpu and gpu-to-cpu transfer rates. Table 1 gives the result of the experiments conducted on the two tablets with the two generations of Qualcomm SoCs. In the table, the notation c-to-g refers to a cpu-to-gpu transfer, similarly g-to-c. The results in Table 1 show that the explicit copy actions between the CPU and GPU are very slow.

Table 1 - Measurement of Transfer Rates

Data Type	APQ8064 **(MByte/s) (Adreno 320/Krait 200)**		APQ8084 **(MByte/s) (Adreno 420/Krait 450)**	
	c-to-g	**g-to-c**	**c-to-g**	**g-to-c**
float	1522	264	97	532

However, in the mobile context, with a single SoC connected to global memory as illustrated in Figure 1, there is no separate cpu or gpu memory as both the GPU and the host CPU use the same main memory. This leads one to believe that the cpu-to-gpu copy and gpu-to-cpu copy be can avoided. In order to do that the OpenCL API *clCreateBuffer* was used with the CL_MEM_ALLOC_HOST_PTR flag. This does indeed work, and makes the explicit copy process redundant, making this part of the mobile GP GPU computation very efficient.

B. Global Memory Throughput Measurement

Access to the global external memory in a GPU (shown as main memory in Figure 1) is much slower compared to the constant, local and private memories that are on-chip within the GPU itself. In order to hide long latency of global memory, GPU memory systems have special hardware that attempts to combine small memory transaction from each thread into a single larger memory access when the data addresses are aligned. These hardware units have become more sophisticated, allowing the memory access to be sequential or permuted, as long as they are within an aligned segment of memory. When this happens the only one memory transaction is issued. These are called *coalesced* memory accesses, and they are illustrated in Figure 2. The black square boxes at the bottom represent the individual thread access requests, and because they are aligned and within a single segment of memory, the red rectangle shows them being coalesced into a single access. In the following subsections we measure the impact of this capability on memory throughput.

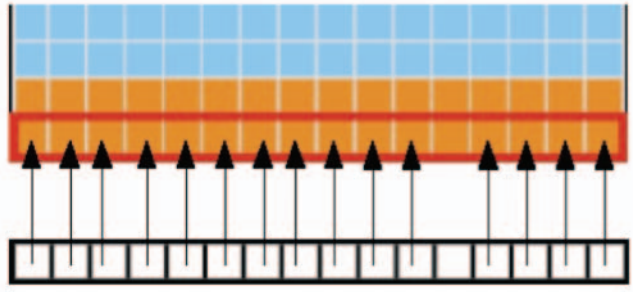

Figure 2 - Coalesced Memory Access[15]

1) Coalesced, Directly Aligned Memory Access

In the first experiment, a single precision floating point array **A** resident in the global memory was copied to another array **B** with same size and data type. This kernel was run with the global *work item size* (which is OpenCL's terminology for number tasks of work to be done) of 67108864; this is equal to the size of the array. The memory allocated in the global memory then becomes 67108864 x 4 bytes = 256MBytes per array. Each work item (when it is spawned in an active thread) makes a single element copy between that same element of **A** and **B** (i.e. **B**[i] = **A**[i]).

The amount of hardware parallelism in the computation can be controlled by setting the number of active threads available in a work group, called the *work group size* in OpenCL terminology. For the Adreno 320 GP GPU (in the 8064 chip) this work group size ranges from 1 to a maximum of 128, and for the Adreno 420 the maximum is 512. This number represents the number of parallel tasks that can be in flight at any given time; note that a smaller number are actually executing in a given specific moment of time. (The latter number is dictated by the amount of parallel hardware, whereas the maximums are likely dictated by task queue sizes in the GPU). To be clear, if the work group size is set to 1, then very little parallelism is possible, as a small portion of the GPU's is being used. When more threads are available, more memory accesses are being made at the same time by the GPU. As described above, we should able to see the ability of the GP GPU to coalesce these aligned, small memory accesses into larger block requests to the external memory, making them far more efficient. In order to show the progressive effect of this coalescing capability, we ran a series of experiments on both GPUs, varying the work group size, from 1 to the maximum, as shown in Figure 3. This figure plots the achieved memory throughput vs. the number of threads available.

The immediate observation from the Figure is that the memory throughput increases significantly as the size of work group increases up to a limit. For the Adreno 320 GPU this limit is roughly when the maximum number of threads is 64 (for a total

throughput of 2 GBytes/s), and for the Adreno 420 the limit is 128 threads and a total throughput of 4 GBytes/s).

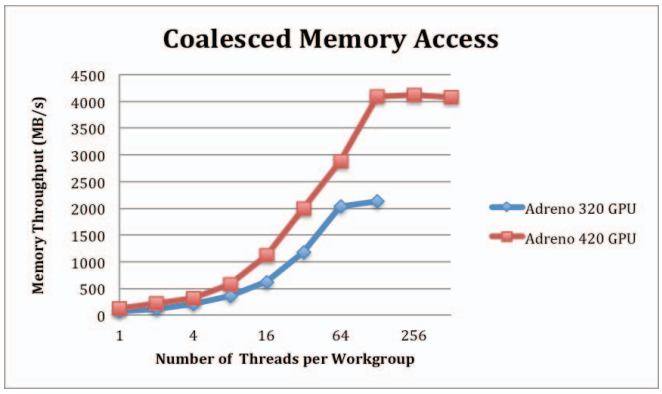

Figure 3 - Memory Throughput for Coalesced Accesses

2) Global Memory Access with Shifts

In the previous section the copy between arrays is perfectly aligned (i.e. **B**[i] = **A**[i]). That makes the coalesced memory accesses align to the natural memory block boundaries of the SoC. In this section we change the copy to have a shift between the two arrays, with a shift amount *k*, and hence the copy code is **B**[i] = **A**[i + *k*]. The shift k in accessing **A** means that a coalesced memory access will cross the internal memory block boundaries and require that the coalesced access be split, into two or more accesses, significantly affecting performance.

Shift sizes of k = 0 to 32 with increments of 1 were used. The kernel execution time was measured for each shift size while keeping workgroup size fixed at the maximum for each GPU – 128 for the Adreno 320 and 512 for Adreno 420. The results of the experiment are depicted in Figure 4. This figure illustrates the effect of alignment – in the few cases where more than one coalesced memory access is needed, throughput is reduced.

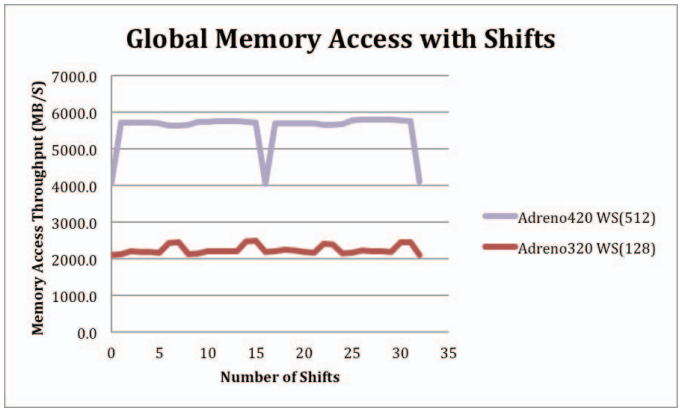

Figure 4 - Memory Access with Shifts

3) Global Memory Access with Strides

Another typical kind of memory access is called *strided* memory access in which array indexes are a constant multiple, *m*, away from each other. To be specific, the array **B** is copied from array **A** in the following manner: **B**[i] = **A**[i * *m*].

As the size of *m* increases, the gap between two accessed memory locations widens significantly which quickly breaches the memory block boundaries. As a result of this, fewer memory accesses are coalesced and more memory transactions will be issued. The same array size was used as before; we also measure the effect of work group size. Figure 5 shows memory throughput as a function of *m* and workgroup size.

The results shown in Figure 5 show that memory accessed with large strides result in many memory transactions hence lower throughput. This is behavior is observed for both GP-GPUs tested. Therefore, this memory access pattern should be avoided. In case it could not be avoided, then optimization techniques such as using the relatively fast local memory of the GPU as a cache should be implemented.

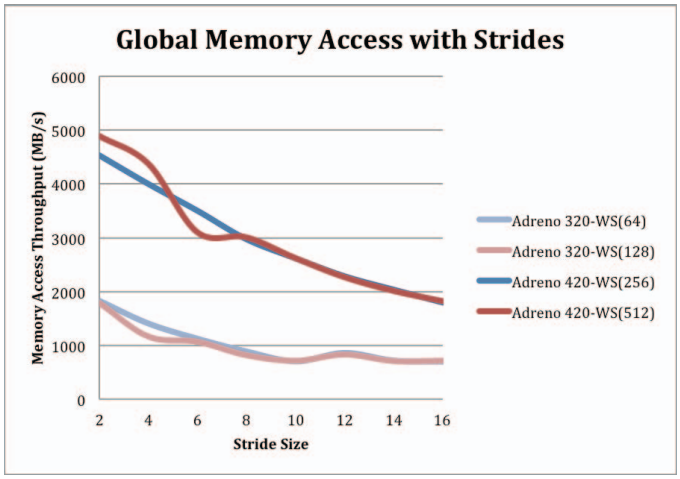

Figure 5 - Global Memory Access with Strides

C. Global Memory Read Latency

The global memory read latency is the time it takes to read an integer from memory; this helps to reveal optimizations in the memory system of the GPU, giving insights into its caching structures. The basic structure of the measurement is to read all elements of an array, and to measure the read time of the accesses. The array size is varied from small (and so likely to fit all inside a cache) to much larger than all the levels of cache. The locality of the accesses is destroyed by making essentially random consecutive accesses, guaranteeing cache misses once the array size is large enough. The code illustrated in Figure 6 is used for this task. It is a version of pointer chasing construct usually used in such measurements [4].

```
__kernel void d_ MeasureMemoryLatency
(__global unsigned int *A, int
dataSize,int iterations)
{
     unsigned int j=0;
     for(int i=0;i<iterations;i++)
     {repeat128(j=A[j];);}
     A[dataSize-1]=j;
}
```

Figure 6 - Code For Memory Latency Measurement

The measurement was done for the CPUs (the Krait 200 and Krait 450) as well as the GPUs for comparison. Figure 7 shows the CPU results and Figure 8 shows the GPU latency.

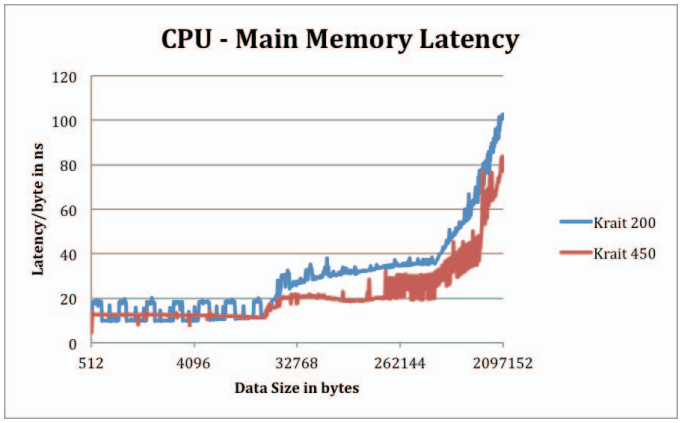

Figure 7 – CPU Main Memory Read Latency

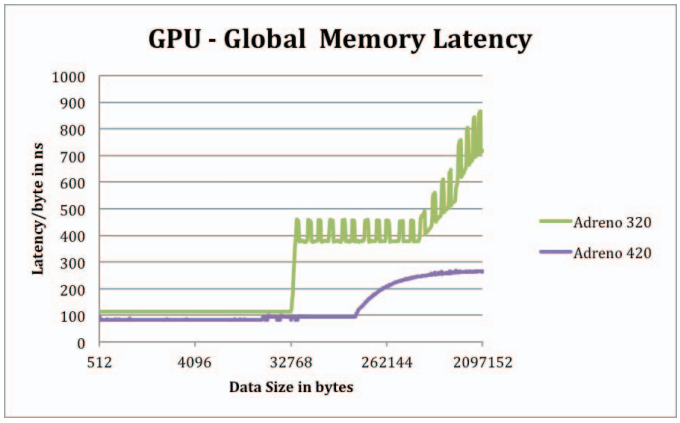

Figure 8 - GPU Global Memory Read Latency

Figure 7 suggests that the Krait 200 and the Krait 450 processors have two levels of cache, and the sizes did not change across the generations: they both have 16 KByte L1 Cache and 512 KByte L2 cache. This can be seen by the jumps in latency measurement at these values on the X-axis.

The GPUs also appear to have two levels of cache; the Adreno 320's L1 is 32Kbytes and a 512Kbyte L2 cache. We suspect that this cache is shared with the Krait 200 CPU, as they are the same size and are quite large. The Adreno 420 L1 cache is smaller, at 16Kbytes, but it contains its own L2 Cache at 128 Kbytes, separate from the Krait 450 CPU's L2 Cache.

The CPUs and the GPUs all share the same global memory, as illustrated in Figure 1, but the latency to global memory is not the same. For example, the Adreno 320 global memory latency, shown at the right side of Figure 8 is approximately 859ns, whereas the CPU global memory latency is about 100ns. These results point to the need to use the Global Memory sparingly or find ways to minimize the latency penalty associated with it.

D. Arithmetic Operation Latency

The actual computation speed of the GPU is crucial in achieving its high performance. In this experiment, the latency of the basic arithmetic operations (+, -, *, /) is measured for the integer and floating-point data types. The measurement is done following the algorithm shown in Figure 9. Two variables, **a** and **b**, are instantiated in the private memory (registers) of the GPU to reduce the memory access latency. The result of the operation is computed in **a** in a cumulative way so that the instructions run are processor bound. That is, since **a** and **b** are in registers, it is assumed **a** is moved to the accumulator once and the value of **b** is operated on **a** repeatedly as shown. Also the kernels are compiled with all compiler optimizations disabled. One other thing to note is that the kernels are run on a single processing element on the GPU. The same code was run for the CPUs (Krait 450 and Krait 200) as well for comparison. The code for the CPU was compiled with the default GCC optimizations.

```
__kernel void
d_measureArithmetic[DataType][Operator]
(int iterations,__global float *result)
{
     DataType a=Const1;
     DataType b=Const2;
     for(int i=0;i<iterations;i++)
     {
       repeat128(a=a[Operator]b;);
     }
result[0]=a;
}
```

Figure 9 - Code for Arithmetic Latency Measurement

This measurement was run on both systems. Table 2 and Table 3 show the time it takes to complete a single operation for integer and floating-point data types in nano seconds. It can be seen that the newer GPU i.e Adreno 420 has superior integer and floating-point performance when compared to the Adreno 320 GPU. Interestingly the GPU's floating-point performance and integer arithmetic performance are very similar. For both integer and floating-point, addition and subtraction operations were the fastest followed by multiplication. The division operation consistently took a significantly longer time in all cases except on the Krait 200 Integer arithmetic. Repeated Integer divisions quickly reach a zero value and all subsequent divisions will have a zero nominator. This leads one to believe that there exists a clever hardware optimization that removes the further division operations in the case of the Krait 200 CPU. Notice that the CPU is more than 10 times faster than the GPU, so the amount of parallelism used in the GPU must be enough to overcome that difference to make using it worthwhile.

Table 2 – Integer Arithmetic Latency Measurements

Integer	Adreno 320(ns)	Krait 200(ns)	Adreno 420(ns)	Krait 450(ns)
Addition	142	29	86	8
Subtraction	143	28	86	8
Multiplication	180	21	94	8
Division	737	12	315	13

Table 3 – Floating Point Arithmetic Latency Measurements

Float	Adreno 320(ns)	Krait 200(ns)	Adreno 420(ns)	Krait 450(ns)
Addition	142	12	86	14
Subtraction	142	12	86	15
Multiplication	143	20	86	15
Division	398	70	176	23

V. Conclusion

We have shown various performance measurements of the memory and computational systems of two modern Mobile GP GPUs. It is interesting to see that because the CPU and GPU are integrated into a single chip, they can access the same main memory, eliminating the need for transfers between these two that are required in desktop/server CPU/GPU systems. The second result shows the impact of coordinating memory accesses on effective memory bandwidth, and it is significant. We determine the size and nature of the cache hierarchies that were not apparent in any of the specifications of the SoCs. In addition, the arithmetic operations latency was measured in the CPU and GPU; while the CPU is significantly faster, the GPU makes up for this with much larger parallelism. It is important to know the difference in speed in order to gauge at what level parallelism there would be a benefit to using the GPU. It was also clear from our experiments that the 8084 SoC has better performance than its predecessor. We believe the insights gained in this work will be useful in efficiently programming these devices. We plan to do this in an upcoming implementation of a computer-vision object detector.

References

[1] Owens, John D., David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. "A Survey of general-purpose computation on graphics hardware." In *Computer graphics forum*, vol. 26, no. 1, pp. 80-113. Blackwell Publishing Ltd, 2007.

[2] Nvidia, C. U. D. A. "Programming guide." (2008).

[3] Khronos OpenCL Working Group, "The OpenCL Specification", Version 2.0, 2014

[4] Wong, Henry, M-M. Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. "Demystifying GPU microarchitecture through microbenchmarking." In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pp. 235-246. IEEE, 2010.

[5] Qualcomm Inc., Snapdragon S4 Processors: System on Chip Solutions for a New Moble Age, White paper, 2011

[6] Qualcomm Inc., Snapdragon 805 Processor. https://www.qualcomm.com/products/snapdragon/processors/805 (accessed April 17, 2014)

[7] Ehringer, David. "The dalvik virtual machine architecture." Techn. report (March 2010) (2010).

[8] Arm Ltd., Introducing NEON™ Development Article, 2009

[9] Pulli, Kari, Wei-Chao Chen, Natasha Gelfand, Radek Grzeszczuk, Marius Tico, Ramakrishna Vedantham, Xianglin Wang, and Yingen Xiong. "Mobile visual computing." In Ubiquitous Virtual Reality, 2009. ISUVR'09. International Symposium on, pp. 3-6. IEEE, 2009.

[10] Cheng, Kwang-Ting, and Yi-Chu Wang. "Using mobile GPU for general-purpose computing–a case study of face recognition on smartphones." In VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on, pp. 1-4. IEEE, 2011.

[11] Rister, Blaine, Guohui Wang, Michael Wu, and Joseph R. Cavallaro. "A fast and efficient SIFT detector using the mobile GPU." In Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on, pp. 2674-2678. IEEE, 2013.

[12] Lowe, David G. "Object recognition from local scale-invariant features." InComputer vision, 1999. The proceedings of the seventh IEEE international conference on, vol. 2, pp. 1150-1157. Ieee, 1999.

[13] Wang, Guohui, Blaine Rister, and Joseph R. Cavallaro. "Workload analysis and efficient OpenCL-based implementation of SIFT algorithm on a smartphone." InProceedings in IEEE global conference signal and information processing (GlobalSIP), pp. 759-762. 2013.

[14] Wang, Guohui, Yingen Xiong, Jay Yun, and Joseph R. Cavallaro. "Accelerating computer vision algorithms using OpenCL framework on the mobile GPU-a case study." In Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on, pp. 2629-2633. IEEE, 2013.

[15] MDP Tablet based on the Qualcomm® Snapdragon™ 805 Processor by Qualcomm Technologies, Inc., http://www.intrinsyc.com/snapdragon-development-platforms/mdp-805-tablet/,(accessed April 19, 2014)