Vasculature segmentation using parallel multi-hypothesis template tracking on heterogeneous platforms

Dong Ping Zhang[1], Lee Howes[2]

[1]Research Group, Advanced Micro Devices, USA
[2]Heterogeneous System Architecture Group, Advanced Micro Devices, USA

ABSTRACT

We present a parallel multi-hypothesis template tracking algorithm on heterogeneous platforms using a layered dispatch programming model. The contributions of this work are: an architecture-specific optimised solution for vasculature structure enhancement, an approach to segment the vascular lumen network from volumetric CTA images and a layered dispatch programming model to free the developers from hand-crafting mappings to particularly constrained execution domains on high throughput architecture. This abstraction is demonstrated through a vasculature segmentation application and can also be applied in other real-world applications.

Current GPGPU programming models define a grouping concept which may lead to poorly scoped local/shared memory regions and an inconvenient approach to projecting complicated iterations spaces. To improve on this situation, we propose a simpler and more flexible programming model that leads to easier computation projections and hence a more convenient mapping of the same algorithm to a wide range of architectures.

We first present an optimised image enhancement solution step-by-step, then solve a separable nonlinear least squares problem using a parallel Levenberg-Marquardt algorithm for template matching, and perform the energy efficiency analysis and performance comparison on a variety of platforms, including multi-core CPUs, discrete GPUs and APUs. We propose and discuss the efficiency of a layered-dispatch programming abstraction for mapping algorithms onto heterogeneous architectures.

Keywords: Image Enhancement, Cardiac Imaging, GPU computing, Programming Model

1. OVERVIEW

We use a real-world application, vasculature image enhancement, as an example to walk through the steps from migrating a single-threaded application to one that utilizes the GPU and APU power using OpenCL. In the following sections, we will first discuss the algorithms to give a background to the problem being solved. Well then show how the CPU implementation may be ported to OpenCL and run on a GPU. During this section well show how different tools may be used to inform the developer about what parts of the application should move to OpenCL and how to optimize them once they are there. Well examine some tradeoffs in kernel code and how they affect performance. Finally well see how these changes may affect energy use by the application and show that the GPU can give an energy consumption benefit.

Beyond that, we use a mathematical tubular model to represent a perfect vessel segment with center position, direction and radius information. To map this model to the local vascular image region, we construct a separable nonlinear least squares problem with linear image parameters and nonlinear model parameters to be solved using a parallel Levenberg-Marquardt algorithm. We examine the underlying programming model of current GPGPU devices for performing template matching, and we propose a new layered approach with improved programmability and better performance portability.

2. VASCULATURE ENHANCEMENT ALGORITHM

The algorithm chosen here is a coarse segmentation of the coronary arteries in CT images based on a multi-scale Hessian-based vessel enhancement filter.[1] The filter utilizes the 2nd-order derivatives of the image intensity after smoothing (using a Gaussian kernel) at multiple scales to identify bright tubular-like structures. The six

second-order derivatives of the Hessian matrix at each voxel can be either computed by convolving the image with second-order Gaussian derivatives at preselected scale value, or approximated using a finite difference approach.

Various vessel enhancement techniques have been proposed in the last two decades. Three of the most popular techniques for curvilinear structure filtering are by Frangi et al.,[1] Sato et al.,[2] and Lorenz et al.[3] All of these approaches are based on extracting information from the second order intensity derivatives at multiple scales to identify local structures in the images. Based on that information it is possible to classify the local intensity structure as tubular-like, plane-like or block-like.

In this work, we use a multi-scale Hessian-based vessel enhancement filter proposed in Frangi et al.[1] because of its superior performance compared with other tubular filters.[4] The filter utilizes the 2nd-order derivatives of the image intensity after smoothing using a Gaussian kernel at multiple scales to identify bright tubular like structures with various diameters. The six second-order derivatives of the Hessian matrix at each voxel are computed by convolving the image with second-order Gaussian derivatives at pre-selected scales. For each voxel, a vesselness metric is computed from the eigenvalues and eigenvectors of its corresponding Hessian matrix. This vesselness reflects the likelihood of a voxel being part of a vasculature network. Hence the algorithm has four major components: Gaussian convolution, Hessian matrix computation, eigen decomposition and sorting, vesselness computation.

3. PERFORMANCE OPTIMISATION OF VASCULATURE ENHANCEMENT

In this section, we present how some of the profiling techniques supported by AMD CodeAnalyst, Profiler, GDEBugger and KernelAnalyser can be used to investigate bottlenecks and improve peak performance of an application. As an example, we port a vasculature enhancement analysis pipeline from a traditional CPU multithreaded execution and optimized for execution in OpenCL on a GPU. We use static analysis and dynamic profiling, evaluate the tradeoffs involved in optimizing a real application for data-parallel execution on a GPU.

We present a vasculature image enhancement module, which is the first and most important step for a template-based vessel extraction application. The automatic and real-time enhancement of the vessels potentially facilitates diagnosis and later treatment of vascular diseases. The performance and power consumption of the proposed algorithms are evaluated on single-core CPU, multi-core CPU, discrete GPU and finally on an Accelerated Processing Unit (APU).

3.1 Hotspot Analysis

Before implementing and optimizing for GPU and APU platforms, we first need to identify the hot spots in the multi-threaded CPU based implementation with the time-based profiling facility in CodeAnalyst. These hot spots are the most time-consuming parts of a program that are the best candidates for optimization. Here we focus on getting an overall assessment of the performance of this application and identifying the hot spots for further investigation and optimization purpose. Hence two profiling configurations suit our requirement: access performance and time-based profiling (TBP).

TBP uses statistical sampling to collect and build a program profile. CodeAnalyst configures a timer that periodically interrupts the program executing on a processor core using the standard operating system interrupt mechanisms. When a timer interrupt occurs, a sample is created and stored for post-processing. Post-processing builds up an event histogram, showing a summary of what the system and its software components were doing. The most time-consuming parts of a program will have the most samples since there would be more timer interrupts generated and more samples taken in that region. It is also important to collect enough samples to draw a statistically meaningful conclusion of the program behavior and to reduce the chance of features being missed entirely.

The System configuration for this analysis is an AMD Phenom II X6 1090T, Radeon HD 6970, with CodeAnalyst Performance Analyzer for Windows version 3.5. The access performance configuration shows 95% average system CPU utilization and 44% average system memory utilization. Given that TBP provides a system-wide profiling, to use efficiently the information provided we select the entries corresponding to the application itself

Component	Convolution	Hessian	Eigenanalysis	Vesselness
Percentage	30%	8%	55%	2%

Table 1. Hotspot analysis of vasculature enhancement application

and perform post-analysis. The Table 1 below provides the four most time-consuming segments of the application, accounting for 95% of the application run-time. It also illustrates the percentage of function execution over the execution time of the whole application. Given that all of these routines are inherently parallel for an image analysis workload, we can start by porting the EigenAnalysis function into an OpenCL kernel first, followed by the convolution, Hessian and vesselness computations.

3.2 Kernel Development and Static Analysis

Here we use the latest release of AMD APP KernelAnalyser, version 1.12. KernelAnalyzer is a tool for analyzing the performance of OpenCL kernels for AMD Radeon Graphics cards. It compiles, analyses and disassembles the OpenCL kernel for multiple GPU targets and estimates the kernel performance without having to run the application on actual hardware or, indeed, having the target hardware in your machine. You can interactively tune the OpenCL kernel using its GUI. It is very helpful for prototyping OpenCL kernels in KernelAnalyser and seeing in advance what compilation errors and warnings would be generated by the OpenCL runtime and subsequently for inspecting the statistics derived by analyzing the generated ISA code.

After developing all four major components from this application in OpenCL and having it tested on the GPU device, we inspect the performance of this newly migrated application in APP Profiler. Figure 1 below shows the timeline view from APP Profiler. From the collected trace file, we can derive that the kernel execution takes 35.1% of the total run-time, the data transfer 32.7%, launch latency 9%, and other unaccounted activities that cover the setup time and finalization time on the host side account for the rest. We can see from the second to last row of the trace that considerable time is spent copying data to and from the device. Given that all four functions are executed on device side and no host-side computation is left in between the kernels, we can safely eliminate the all interim data copies. This optimization reduces the total run-time by 23.4%. The next step is to inspect individual kernels and optimize them.

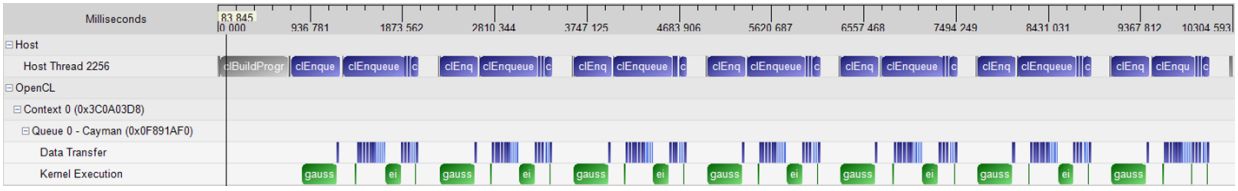

Figure 1. APP profiler timeline view

3.3 Performance Optimisation

In this subsection, we present multiple approaches to increase performance by the evaluating impact of the kernel occupancy ratio, work group size, vector general purpose register count and local data share.

APP Profiler has two main functionalities: collecting application trace and GPU performance counters. Counter selections include three categories: General (wavefronts, ALUInsts, FetchInsts, WriteInsts, ALUBusy, ALUFetchRatio, ALUPacking); GlobalMemory (Fetchsize, cachehit, fetchUnitBusy, fetchUnitStalled, WriteUnitStalled, FastPath, CompletePath, pathUtilisation), LocalMemory (LDSFetchInsts, LDSWriteInsts, LDSBankConflict). For more detailed information about each of these counters, we refer to profiler documentation.

Here we focus on how to use kernel occupancy and a subset of GPU performance counters to guide the optimization. Kernel occupancy is a measure of the utilization of the resources of a compute unit on a GPU. The utilization is measured by the number of in-flight wavefronts (vector threads) for a given kernel, relative to the number of wavefronts that could be launched given the ideal kernel dispatch configuration depending on

the work-group size and resource utilization of the kernel. The kernel occupancy value estimates the number of in-flight (active) wavefronts N_w^A on a compute unit as a percentage of the theoretical maximum number of wavefronts N_w^T that the compute unit can execute concurrently. Hence, the basic definition of the occupancy (O) is given by:

$$O = \frac{N_w^A}{N_w^T} \tag{1}$$

The number of wavefronts that are scheduled when a kernel is dispatched is constrained by three significant factors: the number of general purpose registers (GPR) required by each work-item, the amount of shared memory (LDS for local data share) used by each work-group, and the specified work-group size.

Ideally, the number of wavefronts that can be scheduled corresponds to the maximum number of wavefronts supported by the compute unit because this offers the best chance of covering memory latency using thread switching. However, because the resources on a given compute unit are fixed and GPRs and LDS are hence shared among work-groups, resource limitations may lead to lower utilization. A work-group consists of a collection of work items that make use of a common block of local data storage (LDS) that is shared among the members of the work-group, each consisting of one or more wavefronts. Thus, the total number of wavefronts that can be launched on a compute unit is also constrained by the number of work-groups as this must correspond to an integral number of work-groups, even if the compute unit has capacity for additional wavefronts.

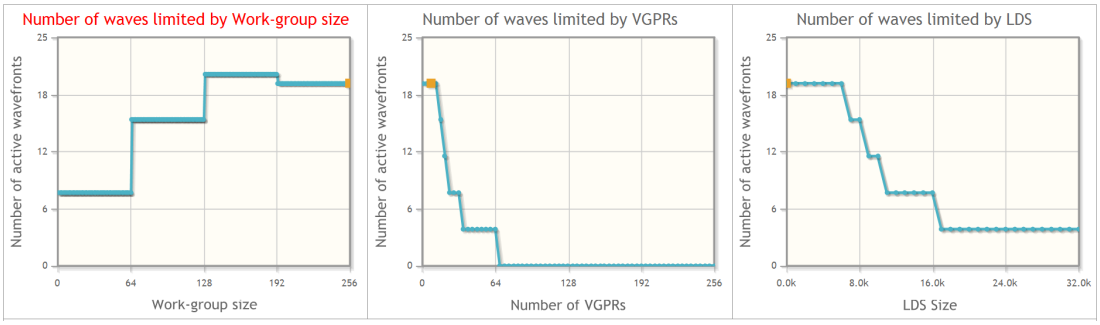

Figure 2. A visualisation of the number of wavefronts on a compute unit as limited by: A) work-group size, B) vector GPRs, C) LDS. This figure is generated by the AMD APP Profiler tool. The highlight of the title shows that the workgroup size is the limiting factor in this profile.

These three graphs in Figure 2 provide a visual indication of how kernel resources affect the theoretical number of in-flight wavefronts on a compute unit. This figure is generated for the convolution kernel with workgroup size of 256 and 20 wavefronts on a Cayman GPU. Given this analysis provided by the kernel occupancy view, the limiting factors would be the optimization target.

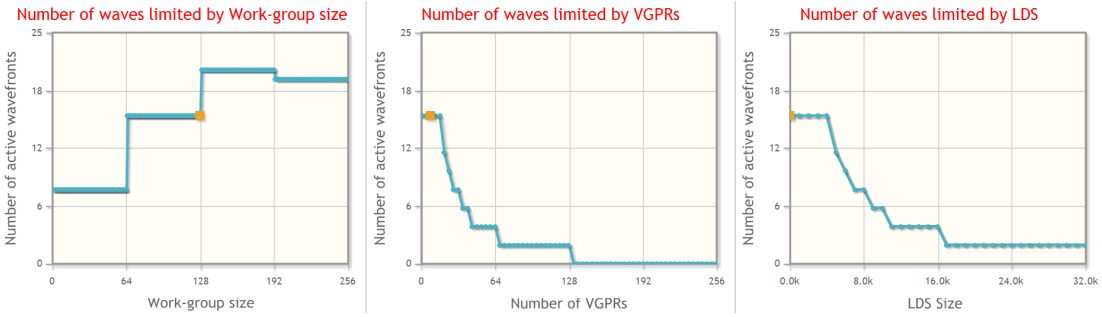

Figure 3. The same visualisation as in Figure 2 with workgroup size lowered to 128. All three factors limit the occupancy.

For illustration purpose, we lower the workgroup size to 128 instead of 192 to check whether we eliminate workgroup size as the limiting factor. A large workgroup size may not, after all, be necessary if enough wavefronts are present to cover memory latency. After this modification, we obtain the a new set of kernel occupancy information as shown Figure 3, where the small yellow square marks the current configuration with workgroup size 128 and wavefronts 16. This change has a negative impact on the occupancy and all three factors are now limiting the number of active wavefronts. However, a high kernel occupancy ratio does not necessary indicate a more efficient execution. Take the convolution kernel as an example. The occupancy rate increases from 76.19% to 95.24% while we increase the workgroup size from {64, 2, 1} to {64, 4, 1}. But through collecting GPU performance counter data, we obtain that the cache hit and ALU utilization ratios reduced from 91.25% and 21.64% to 82.91% and 20.98% respectively. As a result the kernel execution time increases from 342 ms to 353 ms, shown in Figure 4 created with data collected through the GPU performance counter:

Kernel	Platform	GlobalWorkSize			WorkGroupSize			Time	LDS	VGPRs	SGPRs	Scratch Regs	FCStacks	ALU Busy	ALU FetchRatio	Cache Hit	FetchUnit Busy	FetchUnit Stalled
Convolution	Cayman	{ 256	256	200}	{ 64	2	1}	342	0	8	NA	0		521.64	3.72	91.25	46.52	0
Convolution	Cayman	{ 256	256	200}	{ 64	4	1}	353	0	8	NA	0		520.98	3.72	82.91	45.09	0

Figure 4. The impact of workgroup size on performance of convolution kernel

If a kernel is limited by register usage and not by LDS usage then it is possible that moving some data into LDS will shift that limit. If a balance can be found between LDS use and registers such that register data is moved into LDS in such a way that the register count is lowered but LDS does not become a more severe limit on occupancy then this could be a winning strategy. The same may be true of moving data into global memory.

4. MULTIPLE-HYPOTHESIS TEMPLATE TRACKING

4.1 GPGPU programming model

Current GPGPU programming models have inherited NVIDIA's CUDA group-based programming model that stems largely from the way graphics architectures were designed. These models express algorithms as parallel computations with localized communication. They explicitly launch groups of so-called work items (sometimes slightly confusingly known as "threads") that define a more or less static abstract mapping to the architecture. Work items within a single group can communicate with each other. The programmer may apply synchronization primitives, share memory structures and control visibility of operations that are logically scoped to the group.

The problem comes either when the developer attempts to reduce communication overhead by tuning workgroup sizes for the architecture, or when a developer maps an algorithm with tight communication constraints that does not fit any particular workgroup size onto the execution model, resulting in a loss of efficiency.

Unfortunately, mapping complex iteration spaces directly to this sort of grouped launch imposes a severe loss of information. AMD GPUs, for example, are largely based on a 64-element hardware vector (strictly 16 wide SIMD unit pipelined over 4 clock cycles). Synchronization and communication within this single vector is most efficient because it executes as a single thread, with a single program counter. As a resule of this architectural detail, mapping, say, a 15x15x15 region into 64 work items will work well on AMD hardware. However, other architectures do not share this precise detail. NVIDIA's GPUs tend to use a 32-element vector unit, AVX on the CPU is 8 elements wide and so on. To obtain high performance on all of these architectures we would have to re-map the algorithm from 64 work-items to some other number. This remapping requires reinferring certain communication: a process which is likely to be harder than parallelizing the loop in the first place. In addition, projections like this are not clean for the programmer, and understanding the scoping of local memory that is not visible in any sort of lexical scope is difficult.

This architectural mapping problem is one of the issues that makes GPU technology hard to use for people whose expertise is in algorithms development or image processing. We need to be thinking more about mapping

concepts to the underlying architecture, not in mixing expertise such that image processing experts need to also be computer architecture experts.

Instead of attempting to map complicated blocked execution spaces to the GPU we it would be better to start from a clean abstract model. We may program loop nests with a structure such that we have an outer thread/task parallel loop (which could be derived from a loop, from a library call over a lambda, from a task parallel runtime like TBB or similar). Within that loop we have a lexically scoped entity that defines the scope for communication memory and within which scalar/vector code is to execute.

This structure is still not perfect for domain specific areas: the developer still has to map image blocks to tasks and pixels to parallel loops. However, the constructs are more conceptual rather than architectural, and the goal is to bring the abstraction up to a level at which the mapping from image processing constructs to the programming interfaces is more clean.

4.2 Related approaches

OpenCL[5] is the primary focal point of GPU image processing at the moment where cross-platform support is a requirement. OpenCL's mapping from the algorithm to work-items is static, but its mapping of work-items to the architecture is implementation defined. This is where the projection problem may appear.

Intels SPMD compiler[6] attempts to map vector programming onto the CPU and solves some of the cleanliness problems with OpenCL and similar models, getting good efficient in the process. Leissa et al[7] look at portable SIMD programming in a C-like language, explicitly separating scalar from vector code but not using quite the same sort of abstraction.

OpenACC[8] defines mappings of loops using pragmas in an OpenMP-like way. OpenACC's definitions support multiple levels of the mapping but do not heavily structure the algorithm itself to account for this. They merely provide information to the compiler to map the loops that are already there.

4.3 Proposed approach

We propose an abstract work separation such that various high-level languages may separate a rich per-thread but heavily multi-threaded scalar execution flow from code blocks that are explicitly data-parallel, and that may be stored in an intermediate representation for performance-portable mapping to scalar or SIMD architectures. Data parallelism should not be mapped explicitly to SIMD lanes, and to threads as little as possible, but rather be maintained abstractly. SIMD lanes should be an implementation detail.

As an example, lets take a look at a motion-vector search code as we might see compiled for OpenMP:

```
1 #omp parallel for
2 for( int y = 0; y < YMax; ++y ) { // for each macroblock in Y
3  #omp parallel for
4   for( int x = 0; x < XMax; ++x ) { // for each macroblock in X
5     while( not found optimal match ) {
6       // Do something to choose next test region      serial heuristic
7       // Compute tx, ty as coordinates of match region
8       for( int y2 = 0; y2 < 16; ++y2 ) {
9         for( int x2 = 0; x2 < 16; ++x2 ) {
10          diff = block(x2 + 16*x, y2 + 16*y) - target(x2 + tx, y2 + ty);
11        }
12      }
13    }
14  }
15 }
```

In the above code we have two OpenMP parallel loops: these we expect the runtime to map to threads. Mapping the inner loops to threads makes no sense at all, but more than that there is a clear serial while loop in there. If we try to map that while loop into a set of parallel work-items we have to exchange the y2 and

while loops but not just the y2 because that is not enough work-items to fill a wavefront. On an AMD architecture we must exchange the y2 loop and add an x3 loop so that we have an outer x2 loop and an inner x3 loop with the while loop in between. We wrap this in a kernel and rely on a runtime dispatch to handle the outer loops.

```
1 for( int y2 = 0; y2 < 16; ++y2 ) {
2   for( int x2 = 0; x2 < 16; x2 += 4 ) {
3     while( not found optimal match ) {
4       for( int x3 = x2; x3 < x2 + 4; x3 ++ ) {
5         diff = block(x2 + 16*x, y2 + 16*y) - target(x2 + tx, y2 + ty);
6       }
7    }
8 }
```

What we'd end up with as an OpenCL/HSA kernel would be:

```
1 kernel(...) {
2   while( not found optimal match ) {
3     for( int x3 = x2; x3 < x2 + 4; x3 ++ ) {
4       diff = block(x2 + 16*x, y2 + 16*y) - target(x2 + tx, y2 + ty);
5     }
6   }
7 }
```

with a kernel dispatch that did all of:

```
1 for( int y = 0; y < YMax; ++y ) {
2   for( int x = 0; x < XMax; ++x ) {
3     for( int y2 = 0; y2 < 16; ++y2 ) {
4       for( int x2 = 0; x2 < 16; x2 += 4 ) {
```

in one go! Hardly easy to understand. If we have local data in there too the kernel will be something like:

```
1 kernel(...) {
2   local data[...];
3   while( not found optimal match ) {
4     for( int x3 = x2; x3 < x2 + 4; x3 ++ ) {
5       diff = block(x2 + 16*x, y2 + 16*y) - target(x2 + tx, y2 + ty);
6     }
7   }
8 }
```

and the scope of that local data applies to a set of kernel instances for the group.

This is a messy and relatively architecture-specific transformation. Instead we represent what we intend:

```
1 parallelFor( int y = 0; y < YMax; ++y ){ // for each macroblock in Y
2   __attribute(thread_parallel)
3   parallelFor ( int x = 0; x < XMax; ++x ) { // for each macroblock in X
4     local data[...]; // this local data is now lexically scoped to match its
          meaning
5     while( not found optimal match ) {
6       // Do something to choose next test region      serial heuristic
7       // Compute tx, ty as coordinates of match region
8       __attribute(parallel)
9       for( int y2 = 0; y2 < 16; ++y2 ) {
```

```
10          __attribute(parallel)
11          for( int x2 = 0; x2 < 16; ++x2 ) {
12            diff = block(x2 + 16*x, y2 + 16*y) - target(x2 + tx, y2 + ty);
13          }
14        }
15      }
16    }
17  }
```

The inner loops are now marked as parallel, and hence vectorisable. The outer loops are launched using pure library calls and we can now put local scoped data back in. The inner loops can now be mapped down to the appropriate vector size and loop nest to cover that region without undoing the transformation the programmer had to make to map to OpenCL or CUDA.

5. EXPERIMENT AND RESULTS

In this paper, we only present the experiment setting and results from the vasculature enhancement sections. An experimental implementation and evaluation of our newly proposed programming model for easier mapping of template tracking to heterogeneous architecture is the focus of our future work. We tested the optimised enhancement application on an AMD APU (A10-5800K) platform in three formats: a single-threaded CPU version, a multi-threaded CPU version and the APU-based version, as shown in Table 2.

	CU0	CU1	GPU	Power	Execution time (s)	Energy
CPU: Single Thread	23.97	21.52	8.71	54.20	282.00	15285.45
CPU: Multiple Thread	32.24	31.78	9.01	73.03	133.80	9772.01
APU	16.01	15.21	24.48	55.70	19.35	1077.72

Table 2. Performance and energy consumptions on three approaches

Baseline: CPU Multi-thread	Performance Improvement	Energy Improvement
APU	6.9	9

Table 3. Performance and engery improvement on one APU platform

Two CPU modules (core pairs) listed as CU in the table above and GPU. Power cosumption samples are collected at 100ms time intervals and seven different scales for identifying vessel structure are used to both capture the viariation of vascular structure sizes and increase the amount of samples collected for a more accurate analysis. Each row in Table 2 shows the average power consumption for each device and the total power consumption and execution time measured. Using this data we calculate the total energy consumption. Using multi-threaded CPU entry as baseline, we can derive the performance and energy improvement for a APU based one shown in Table 3.

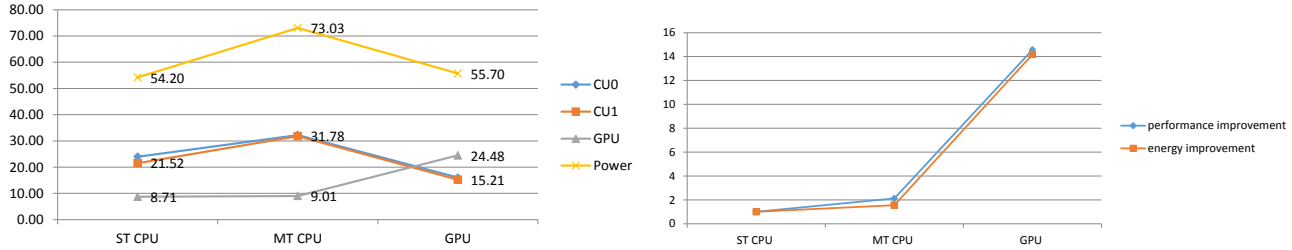

Figure 5. (a): Power consumption of each component; (b): Energy consumption and performance comparison.

In summary, we observe that despite the GPU only being powered down while not in use, the energy consumption of executing the application on APU is significantly lower than the multi-threaded CPU test, as shown in Figure 5.

6. CONCLUSIONS

We presented how to use profiling tools to analyze and hence help optimize kernel performance, and applied runtime profiling and static analysis to an image analysis application to help optimize performance using OpenCL. Here we chose the hardware and tools available for us. However, the techniques proposed can be applied to a broad range of architectures from same vendor or other vendors to achieve similar goals. We also evaluate the efficiency of current GPGPU programming model and propose a layered-dispatch abstraction for mapping algorithms onto heterogeneous architectures that is potentially benefitial to many other applications.

REFERENCES

1. A. Frangi, *Three-dimensional model-based analysis of vascular and cardiac images*. PhD thesis, University Medical Center Utrecht, The Netherlands, 2001.
2. Y. Sato, T. Araki, and M. Hanayama, "A viewpoint determination system for stenosis diagnosis and quantification in coronary angiographic image acquisition," *IEEE Transactions on Medical Imaging* **17**, pp. 121–137, Feb 1998.
3. C. Lorenz, I.-C. Carlsen, T. Buzug, C. Fassnacht, and J. Weese, "Multi-scale line segmentation with automatic estimation of width, contrast and tangential direction in 2D and 3D medical images," in *CVRMed-MRCAS '97: Proceedings of the First Joint Conference on Computer Vision, Virtual Reality and Robotics in Medicine and Medial Robotics and Computer-Assisted Surgery*, pp. 233–242, 1997.
4. S. Olabarriaga, M. Breeuwer, and W. Niessen, "Evaluation of Hessian-based filters to enhance the axis of coronary arteries in CT images," in *Computer Assisted Radiology and Surgery*, *International Congress Series* **1256**, pp. 1191–1196, 2003.
5. OpenCL Working Group, "The OpenCL specification, version 1.2, revision 19.." Khronos, 2012.
6. M. Pharr and W. R. Mark, "ispc: a SPMD compiler for high-performance cpu programming," in *Innovative Parallel Computing Conf.*, May 2012.
7. R. Leissa, S. Hack, and I. Wald, "Extending a c-like language for portable simd programming," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, *PPoPP '12*, pp. 65–74, ACM, (New York, NY, USA), 2012.
8. "The openacc specification." http://www.openacc-standard.org/, 2011.