GPGPU WORKLOAD ANALYSIS AND MEDIA PERFORMANCE STUDIES

by

ASHWINI SIMHA

A PROJECT REPORT

Presented to the Faculty of the Graduate School of the

UNIVERSITY OF WISCONSIN-MADISON

In Partial Fulfillment of Requirements for the Degree

MASTER'S OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

Spring 2011

ABSTRACT

This project was done with the Mobile Microprocessor Group at Intel Corporation as a part of a six month internship.

The primay objective of this project was to study the performance of GPGPUs (General purpose computation on Graphics Processing Units) for various benchmark applications. GPGPUs have gained wide spread importance in recent years because of their extensive computation capabilities. A GPGPU uses the GPU, which traditionally handles computation only for graphics, to perform computation for applications traditionally handled by the central processing unit (CPU). My role was to do detailed performance analysis on silicon to understand various aspects of the micro-architecture, usefulness for future generations; as well as gain some experience and expertise in developing ,programming and optimizing GPGPU workloads.

During the life of the project of six months, initial phase was learning GPGPU programming models, Open Computing Language (OpenCL) architecture, thread management in OpenCL, and a new Intel propriety language used to for media programming called C for Media.

Another topic of widely accepted importance was media performance studies (ongoing). The high level goal of this exercise was to understand and analyze performance bottlenecks in a generic media playback pipeline scenario, optimize it and thus increase efficiency of a media playback pipeline.

ACKNOWLEDGEMENT

Foremost, greatest thanks to my academic advisor Prof. Kewal K. Saluja, who has been an excellent advisor in all aspects. His approachability, patience and immense knowledge have been very valuable through my journey as a Masters student.

I would also like to thank my mentor at Intel India, Dr. Ajay Joshi for his inspiring ideas, encouragement and to have taught me to think outside of the box.

I would like to thanks Prof. Bernard Lesieutre and Prof. Yu Hen Hu for being gracious to have accepted this form of submission as requirement for graduation. Also, would like to thank Sheralynn Liantonio of ECE Student Services for her great patience.

Last but not least, I would also like to thank my parents for always being supportive through all times.

TABLE OF CONTENTS

LIST OF FIGURES Page no

INTRODUCTION TO OPENCL

Parallelism is an important step towards better performance. Central Processing Units (CPUs) improve performance by adding multiple cores and Graphics Processing Units (GPUs) have evolved from fixed function rendering devices into programmable parallel processors. As today's computer systems have highly parallel CPUs, GPUs and other types of processors, it is important to enable software developers to take advantage of these heterogeneous processing platforms.

OpenCL is a portable and efficient standard for parallel programming for CPUs, GPUs and other processor. The Khronos OpenCL Specification[4][5] describes OpenCL to be "suited to play an increasingly significant role in emerging interactive graphics applications that combine general parallel compute algorithms with graphics rendering pipelines. OpenCL consists of an API for coordinating parallel computation across heterogeneous processors; and a cross-platform programming language with a well specified computation environment."

2. PERFORMANCE ANALYSIS OF *NBODY* KERNEL ON ATI AND SANDY BRIDGE

2.1. INTRODUCTION

One of the first applications that we investigated for performance analysis was *NBody* Kernel. In this section we explain the problem, its implementations, and our observations.

2.2. *NBODY* KERNEL

The N-body problem simulates the evolution of a system of *N* bodies, where the force exerted on each body arises due to its interaction with all the other bodies in the system. The algorithm calculates the force which acts on one particle from all the other particles in the system[1][2]. An example of the algorithm is the simulation of the motion of all of the particles (stars) in a system (galaxy), where simulation represents the motion of stars in a galaxy.

The goal of the exercise was to understand and analyze the impact of Shared Local Memory (SLM) [3][4] on the performance of *NBody* kernel on ATI 5550 discrete graphics card and on Sandy Bridge(SNB).

2.3. SHARED LOCAL MEMORY ON GPU

The core of the OpenCL execution model is defined by how the kernels execute. When a kernel is submitted for execution by the host, an *index space* is defined. An instance of the kernel executes for each point in this index space [4]. This kernel instance is called a *work-item* and is identified by its point in the index space, which provides a global ID for the work-item. Each work-item executes the same code but the specific execution pathway through the code and the data operated upon can vary per work-item, similar to SIMD (Single Instruction Multiple Data Architecture). Work-items are organized into *work-groups*. The work-groups provide a more coarse-grained decomposition of the index space. Work-groups are assigned a unique work-group ID with the same dimensionality as the index space used for the work-items. Work-items are assigned a unique local ID within a work-group so that a single work-item can be uniquely identified by its global ID or by a combination of its local ID and work-group ID. The work-items in a given work-group execute concurrently on the processing elements of a single compute unit.

Work-item(s) executing a kernel have access to four distinct memory regions: Global, Constant, Local and Private Memories. Global memory permits read/write access to all work-items in all work-groups. Constant Memory is a region of global memory that remains constant during the kernel execution. Local Memory region is local to a work-group and can be used to allocate variables that are shared by all work-items in that work-group. It may be implemented as dedicated regions of memory on the OpenCL device. Alternatively, the local memory region may be mapped onto sections of the global memory. Private Memory is private to a work-item. Variables defined in one work-item's private memory are not visible to another work-item.

The reason for choosing *NBody* kernel for this implementation is because there is a lot of data reuse in this kernel thus making it a good example to test the usage of shared local memory.

2.4. **OPENCL ATI *NBODY* IMPLEMENTATION**

As explained earlier, there are two main steps in the execution of *NBody* kernel:

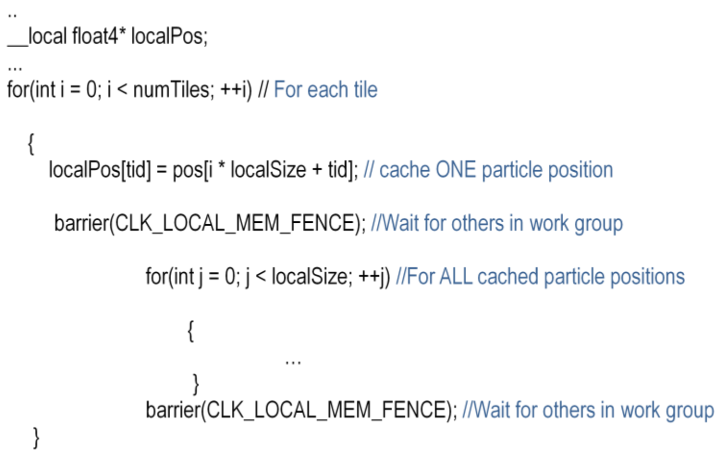
- Calculation of the total force on each particle resulting from the gravitational attraction to all other particles.
- Determination of position and velocity of each particle as result of the force calculated in previous step.

Repeating this process results in a simulation of the motion of all of the particles (stars) within the system (galaxy) [1].

2.5. PRISTINE IMPLEMENTATION (ATI SDK)

The kernel code is invoked once for each *N* particles. Kernel code updates position and velocity of one particle in each invocation. Kernel will be executed in work-groups similar to tiles or blocks of the index-space. Work-items within a work-group are locked together in execution and can share data and work cooperatively. The force on every particle depends on position of all other particles, thus there is an opportunity for data re-use. Shared local memory (SLM) is used to store a block of particle positions to be used by all work-items in a work-group. Each work-item copies one particle position into SLM from Global Memory. The outer loop iterates over work group tiles and the inner loop iterates over a block of particle positions that are assumed to have been cached in SLM. (Since multiple instances of the kernel are executed in parallel, kernel trusts that other work-items did their job).

Comparing ATI and SNB performance on the pristine version, we observed that ATI performance is always better than SNB because ATI takes benefit of the SLM usage.

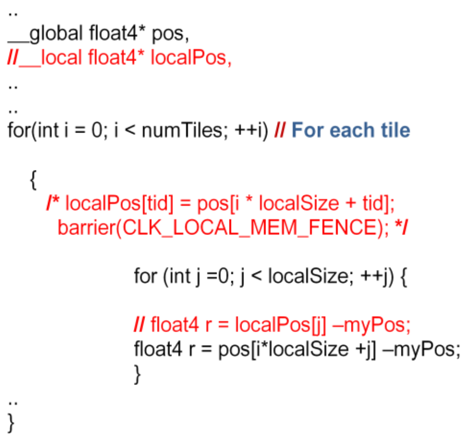


```
..
__local float4* localPos;
...
for(int i = 0; i < numTiles; ++i) // For each tile

   {
      localPos[tid] = pos[i * localSize + tid]; // cache ONE particle position

       barrier(CLK_LOCAL_MEM_FENCE); //Wait for others in work group

                   for(int j = 0; j < localSize; ++j) //For ALL cached particle positions

                         {
                                        ...
                          }
                   barrier(CLK_LOCAL_MEM_FENCE); //Wait for others in work group
    }
```

2.6. MODIFIED IMPLEMENTATIONS

2.6.1. No Shared Local Memory, tile by tile implementation (Part1 Flavor 1)

There is no pointer to SLM since particle positions are not cached for re-use. Particle positions are read directly from global memory. But we still retain the tile by tile access of global memory i.e., we maintain 2 nested loops.



```
..
__global float4* pos,
//__local float4* localPos,
..
..
for(int i = 0; i < numTiles; ++i) // For each tile

   {
     /* localPos[tid] = pos[i * localSize + tid];
       barrier(CLK_LOCAL_MEM_FENCE); */

                  for (int j =0; j < localSize; ++j) {

                  // float4 r = localPos[j] –myPos;
                  float4 r = pos[i*localSize +j] –myPos;
                  }
..
}
```

Observations:

- ATI: For smaller problem sizes (N<1K), the pristine version with SLM performs better. And for problem sizes greater than 1 K, modified version performs better.
- SNB: There is no variation in performance between the two versions i.e., performance of *NBody* kernel on SNB is independent of shared local memory.

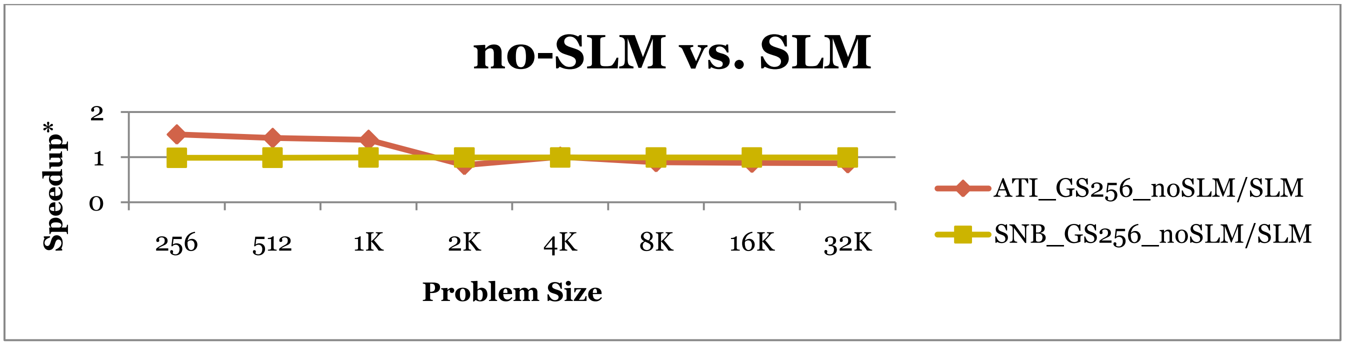


FIG 1. NO-SLM vs. SLM – PART1 FLAVOR 1

2.6.2. Optimizing loops (non-tiled) for no shared local memory (Part1 Flavor2)

There is no pointer to local memory since particle positions are not cached for re-use. Since particle position is read directly from global memory, there is no need for tile by tile access from global memory ie there is only one “for” loop.

```
..
__global float4* pos,
//__local float4* localPos,
int numBodies, // Problem Size
..
/* for(int i = 0; i < numTiles; ++i)
   {
       localPos[tid] = pos[i * localSize + tid];
       barrier(CLK_LOCAL_MEM_FENCE); */

              for (int j =0; j < numBodies; ++j) {

              // float4 r = localPos[j] –myPos;
              float4 r = pos[j] –myPos;
..
}
```

Observations:

We observe similar trends as the previous version. This flavor performs better than the previous flavor since we do not have nested loops and we have optimized the tile-by-tile access to global memory.

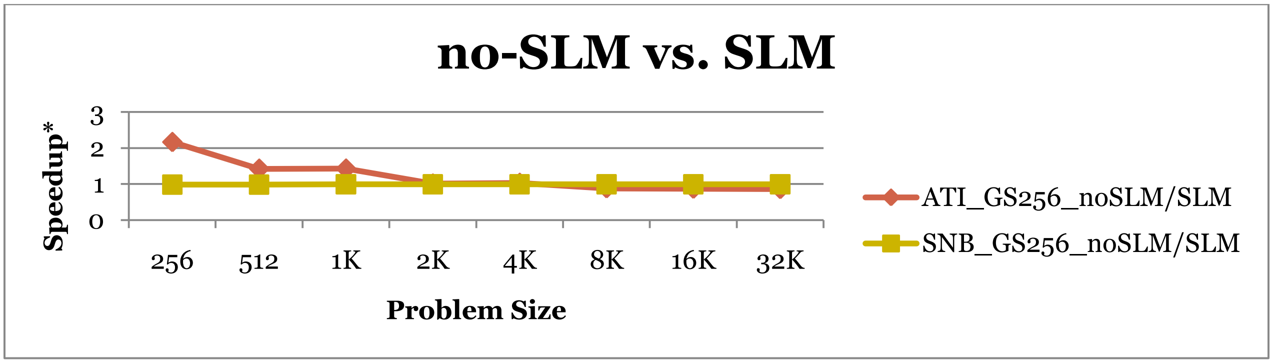
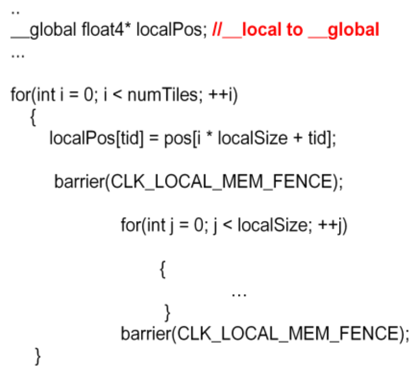


FIG 2. NO-SLM vs. SLM – PART1 FLAVOR 2

2.6.3. Poor Shared Local Memory (Part2)

This simulates shared local memory through global memory. This is clearly a poor implementation since local memory has similar latency as global memory.



```
..
__global float4* localPos; //__local to __global
...

for(int i = 0; i < numTiles; ++i)
  {
     localPos[tid] = pos[i * localSize + tid];

      barrier(CLK_LOCAL_MEM_FENCE);

               for(int j = 0; j < localSize; ++j)

                    {
                               ...
                     }
               barrier(CLK_LOCAL_MEM_FENCE);
  }
```

Observations:

Modified Implementation performs much worse than the pristine version since for every access, it has to reach out to global memory and for every access it faces global memory latencies.

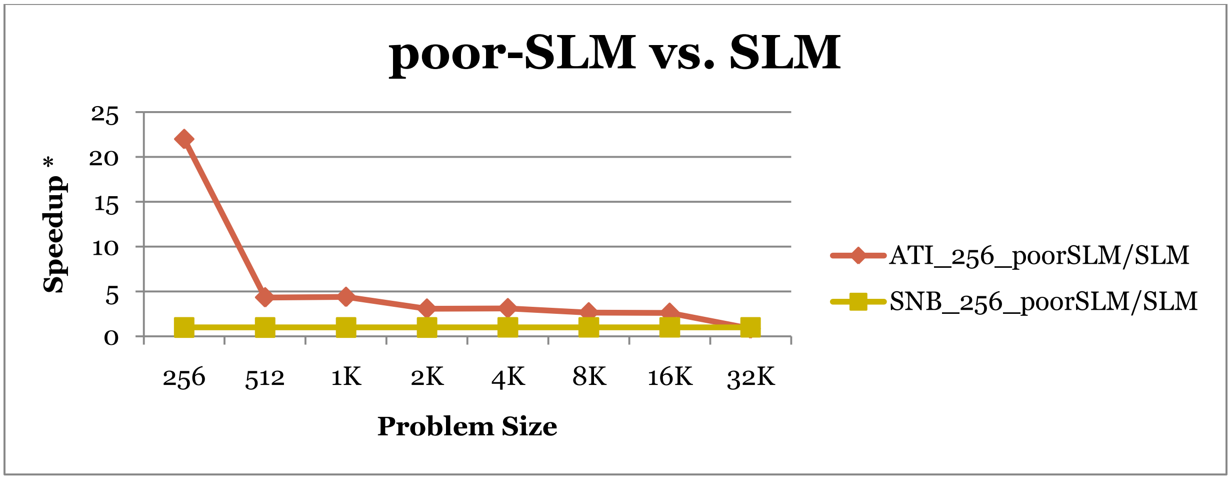


FIG 3. POOR-SLM vs. SLM

2.7. CONCLUSION

- **ATI comparison with SNB:**
  - SNB ~2x slower than ATI for N <1K
  - SNB ~10x slower than ATI for N>1K

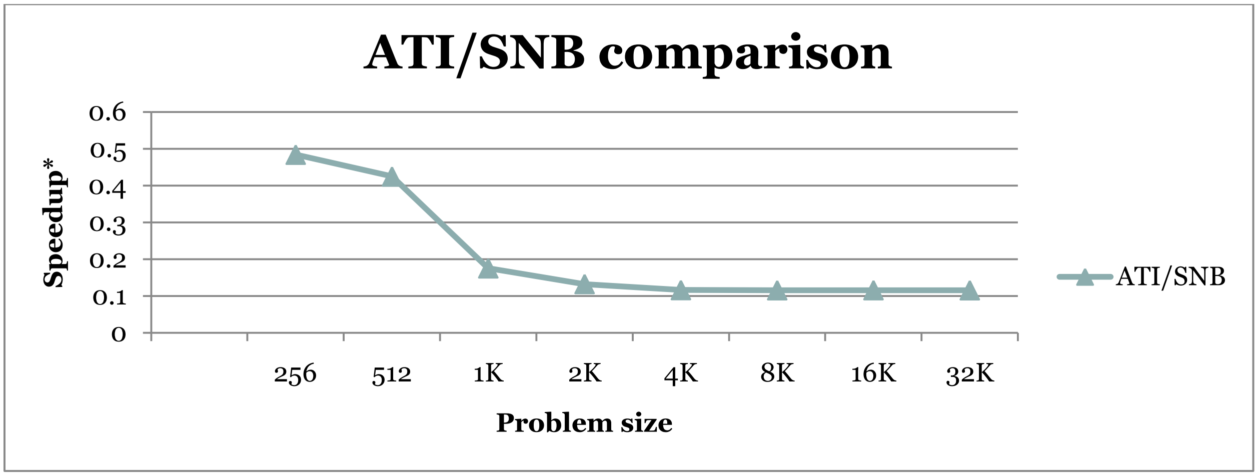


FIG 4. ATI vs. SNB COMPARISON

- **Best Implementation on ATI:**
  - SLM version for N< 2K
  - No SLM with 1 loops (P1F2) N>2K
  - Poor-SLM/SLM has not been plotted, since, from the previous graph, it is clearly much worse than the rest.

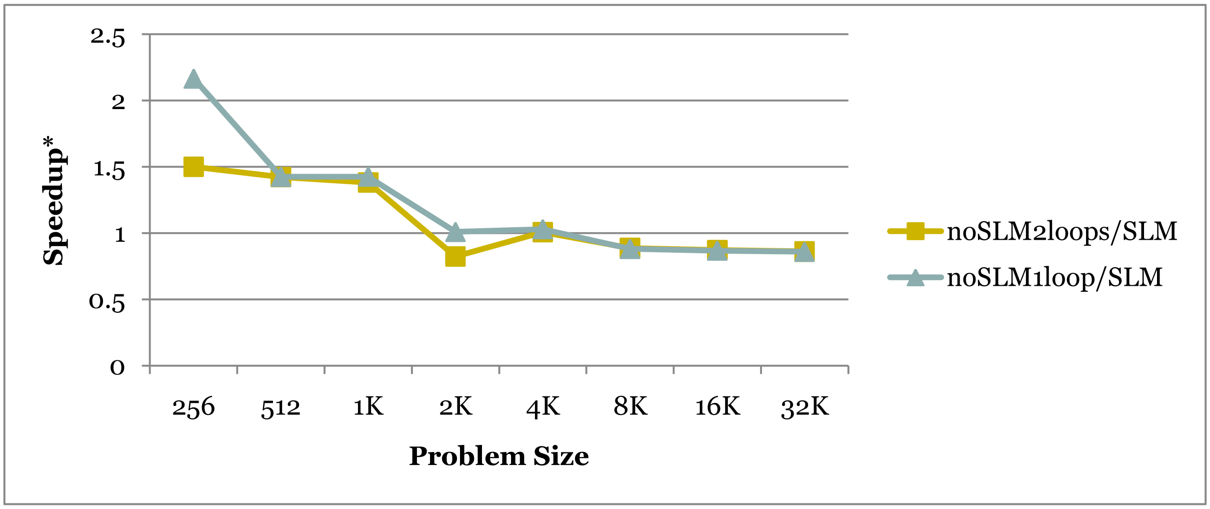


FIG 5. BEST IMPLEMENTATION ON ATI

- **Best Implementation on SNB:**
  - Performance is independent of SLM on SNB i.e. all implementations perform the same

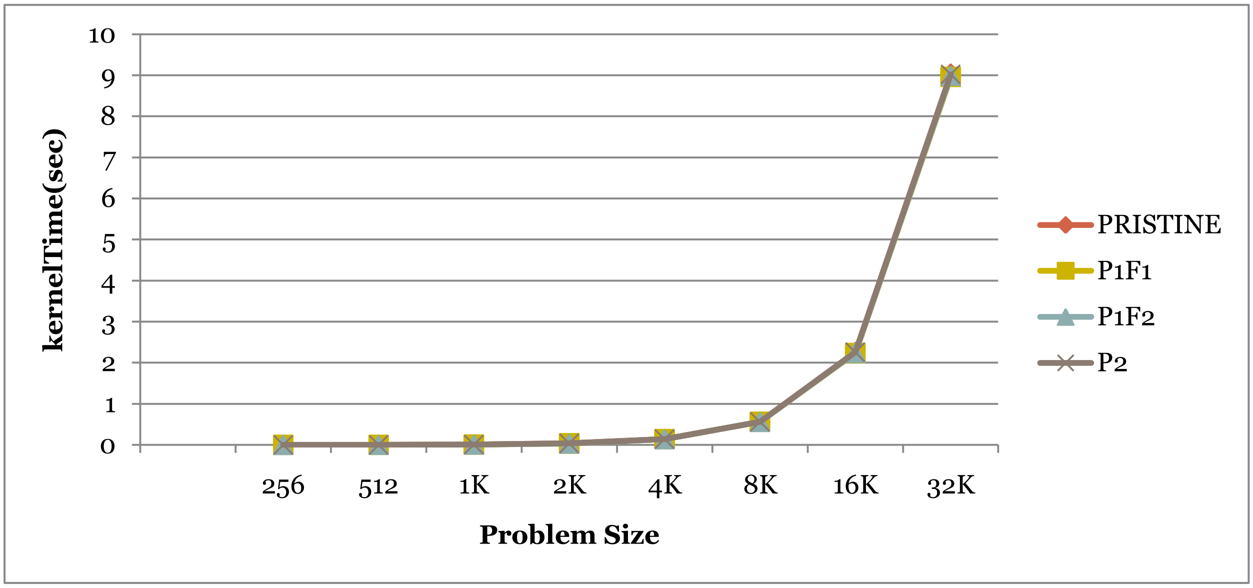


FIG 6. BEST IMPLEMENTATION ON SNB

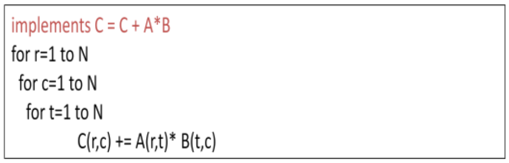
3. PERFORMANCE ANALYSIS OF SINGLE PRECISION GENERAL MATRIX MULTIPLY (*SGEMM)* KERNEL

3.1. INTRODUCTION

This was done on a kernel implementation in C-for-Media which is intended to support high level programming of media kernels[9][10][11]. The goal of this exercise was to understand how the new language compared against OpenCL in performance. The algorithm chosen for this exercise was Single Precision General Matrix Multiply algorithm (SGEMM). I started by understanding the algorithm in detail, reading documents to understand APIs and later took measurements. The important part was to understand how APIs from these two different standards mapped in order to have a fair comparison.

3.2. SGEMM KERNEL

SGEMM stands for Single Precision General Matrix Multiply which performs the operation: C+=A*B. ATI SDK implements *MatrixMultiplication* C=A*B. We had to change ATI SDK to implement C+=A*B. We compared with no_SLM version.



```
implements C = C + A*B
for r=1 to N
 for c=1 to N
  for t=1 to N
         C(r,c) += A(r,t)* B(t,c)
```

3.3. TILING ON MATRIX MULTIPLY

Basic idea of tiling is to rearrange data sets for smaller working sets. This helps increase compute intensity [6]. The blocked algorithm has computational intensity q ~= b where is b is block size. The larger the block size, the more efficient our algorithm will be.

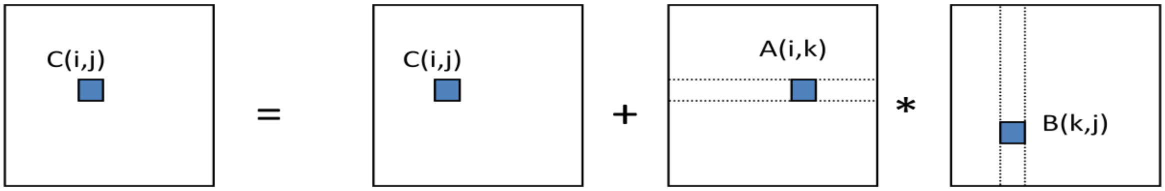
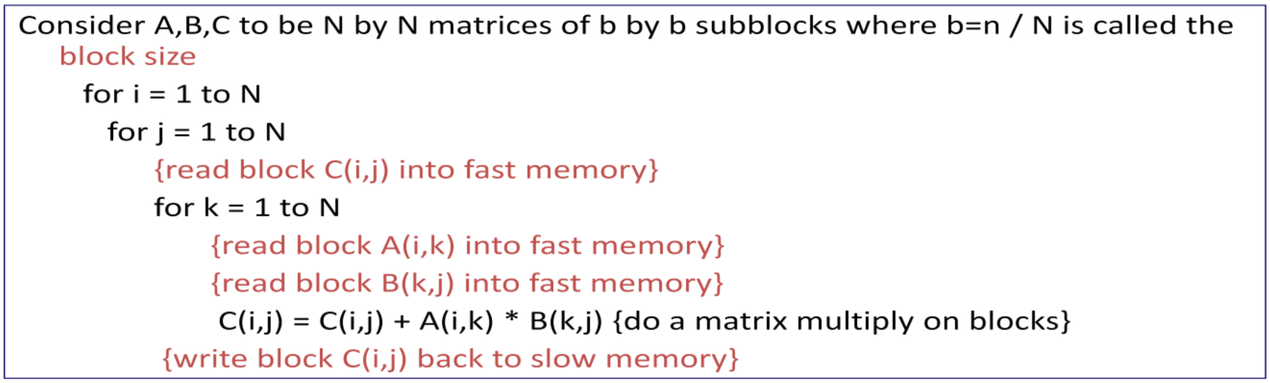


FIG 7. TILING ON MATRIX MULTIPLY



```
Consider A,B,C to be N by N matrices of b by b subblocks where b=n / N is called the
   block size
     for i = 1 to N
       for j = 1 to N
           {read block C(i,j) into fast memory}
           for k = 1 to N
                {read block A(i,k) into fast memory}
                {read block B(k,j) into fast memory}
                 C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
           {write block C(i,j) back to slow memory}
```

FIG 8. TILING ON MATRIX MULTIPLY 2

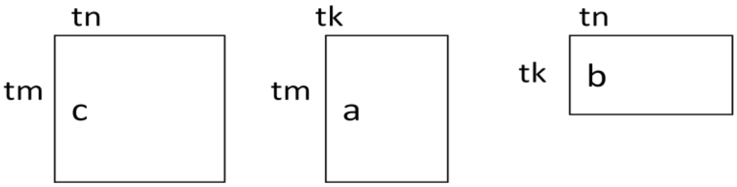
Limit: All three blocks from A,B,C must fit in fast memory (cache), so we cannot make these blocks arbitrarily large. Compute intensity is ≈b for large N. Larger the block, lesser is the bandwidth needed.

3.4. TILE SIZE VARIATION

We varied the sizes of the tiles to see the impact on performance.

1. TILESIZE0: tm = 16; tn = 8; tk = 8;
2. TILESIZE1: tm = 32; tn = 16; tk = 8;
3. TILESIZE2: tm = 64; tn = 32; tk = 16;

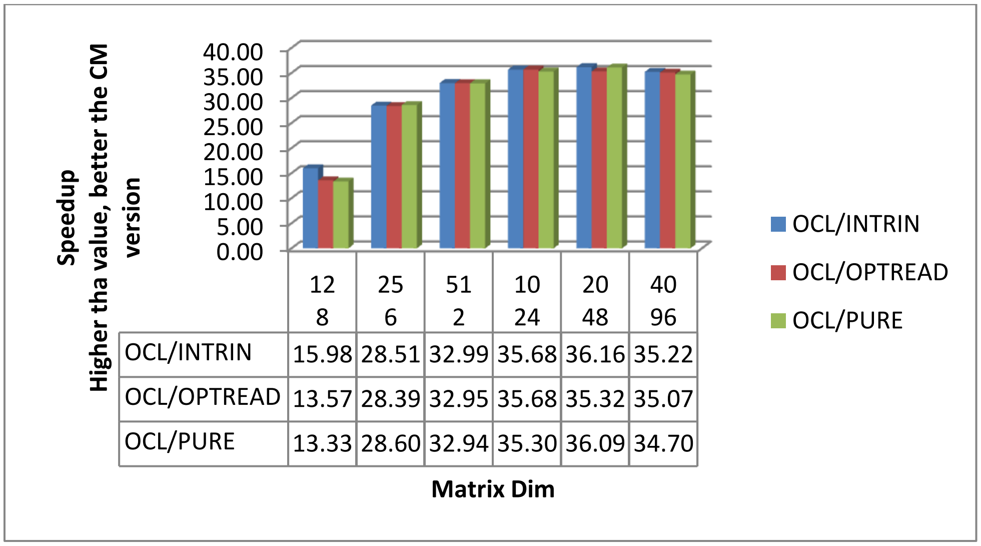
This tile size implementation did not run due to lack of registers. Register allocation failed for this implementation.



Observation:

Optimal Tile Size1: tm = 32; tn = 16; tk =8.

There are 4 versions of the code of SGEMM written in C for Media. We could test run only three versions on our system: Intrinsic (INTRIN), Optimized reads (OPTREAD) and pure (PURE) version. Comparing each of this with the OpenCL version showed the performance difference between the same kernels on the two different platforms.



	128	256	512	1024	2048	4096
OCL/INTRIN	15.98	28.51	32.99	35.68	36.16	35.22
OCL/OPTREAD	13.57	28.39	32.95	35.68	35.32	35.07
OCL/PURE	13.33	28.60	32.94	35.30	36.09	34.70

FIG 9. OPENCL vs. CM COMPARISON

3.5. CONCULSION

- CM kernel performs ~30 x better than OCL version for large matrix dimensions.
- Optimal Tile Size: tm = 32; tn = 16; tk =8.

4. MONTE CARLO EUROPEAN OPTION PRICING IMPLEMENTATION AND OPTIMIZATION

4.1. INTRODUCTION

Having got a fair understanding of C for Media, my next task was to implement Monte Carlo European Option Pricing kernel in this language. Once the implementation was complete, I had to optimize the kernel for best performance. Performance was measured by the number of samples that could be processed per second. We started with detailed understanding of the algorithm. This exercise exposed me to several intricate details of the language and APIs. I worked in collaboration with Satish Nadathur, Misha Smeliyansky and Dhiraj Kalamkar (Intel Labs, Santa Clara) for optimizing the kernel and in this process learnt several best known methods (BKMs) for CM.

4.2. OPTION PRICING

The most common definition of an option is an agreement between two parties, the option seller and the option buyer, whereby the option buyer is granted a right (but not an obligation), secured by the option seller, to carry out some operation (or exercise the option) at some moment in the future [12].

The goal is to find the present day fair price of the option using a set of mathematical equations**.** The first stage of computation is the generation of a normally distributed number sequence which comes down to uniformly distributed sequence generation. Once we've generated the desired number of samples, we use them to compute an expected value and confidence width for the underlying option.

Pricing a single European option using Monte Carlo integration is inherently a one-dimensional problem, but if we are pricing multiple options, we can think of the problem in two dimensions [13]. We represent paths (samples) for an option along the x axis, and options along the y axis. This makes it easy to determine our grid layout: we'll launch a grid X blocks wide by Y blocks tall, where Y is the number of options we are pricing.

The total number of threads was set to product of number of threads in X dimension and number of threads in Y dimension (N= Tx*Ty). Thread 0 (T0) handles Option 0 in chunks of SIMD 8. Then goes to Option(0+N) and so on, ie each thread handles options/total_threads.

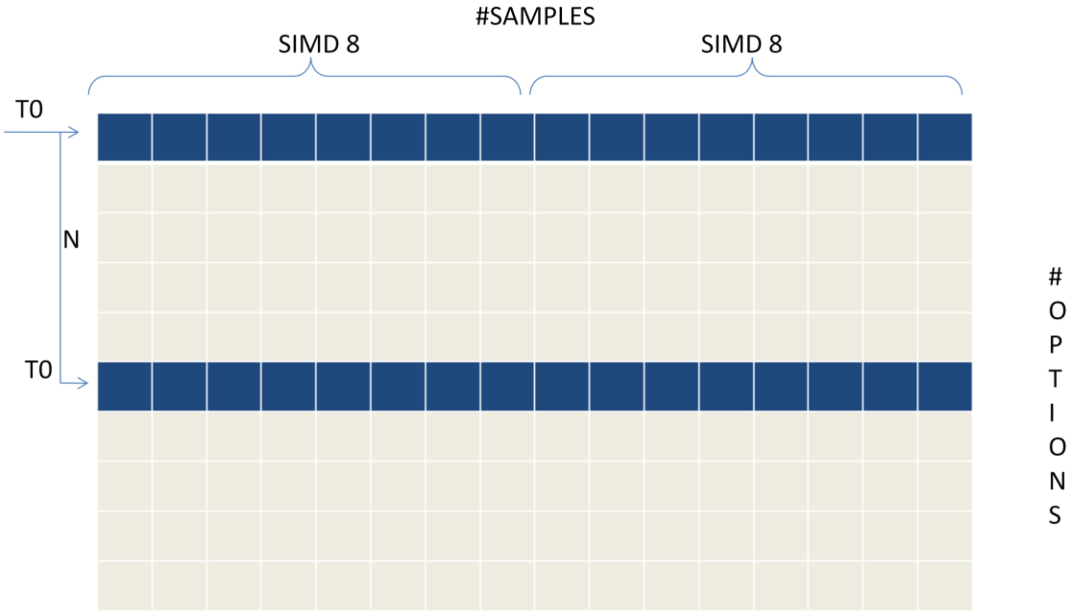


FIG 10. MONTE CARLO GRID

4.3. DETAILS OF IMPLEMENTATION

Measurements:

- Number of threads: X dimension (Tx) = 16; Number of threads: Y dimension (Ty) = 16;
- Number of SAMPLES = 8192; Number of OPTIONS = 4096
- Samples/sec = (#Samples*#Options)/time(sec))

We implemented the baseline version and all the optimizations were done on baseline version.

4.3.1. OPTIMIZATION 1: Optimized Reads:

Reading the inputs (random samples) is done in chunks of 8x8 samples. Performance was ~1.5x over baseline version.

4.3.2. OPTIMIZATION 2: Optimized Read + loop unroll:

Each optimization is done over the previous optimization. In this step, we unrolled innermost loop. Performance was ~2.6x over baseline version.

4.3.3. OPTIMIZATION 3: cm_max() instruction usage:

We are Interested only in positive prices (since if profits are negative, it is considered zero: algorithm specific). Performance improvement ~2.9x over baseline version.

4.3.4. OPTIMIZATION 4: Moving Reduction outside loop:

Reduction of SIMD values to single value is done inside inner most loop. Moving it outside reduces number of instructions in core loop and thus, performance is ~3.1x over baseline.

4.3.5. OPTIMIZATION 5: Tiling Options:

Each thread operates on (OPTION/TOTAL_THREADS) options in contiguous blocks. When random sample values same for all options, multiple options can be processed together, by reuse of random samples. The sweet spot for number of options was 4 options. Performance is ~4.1x over baseline version.

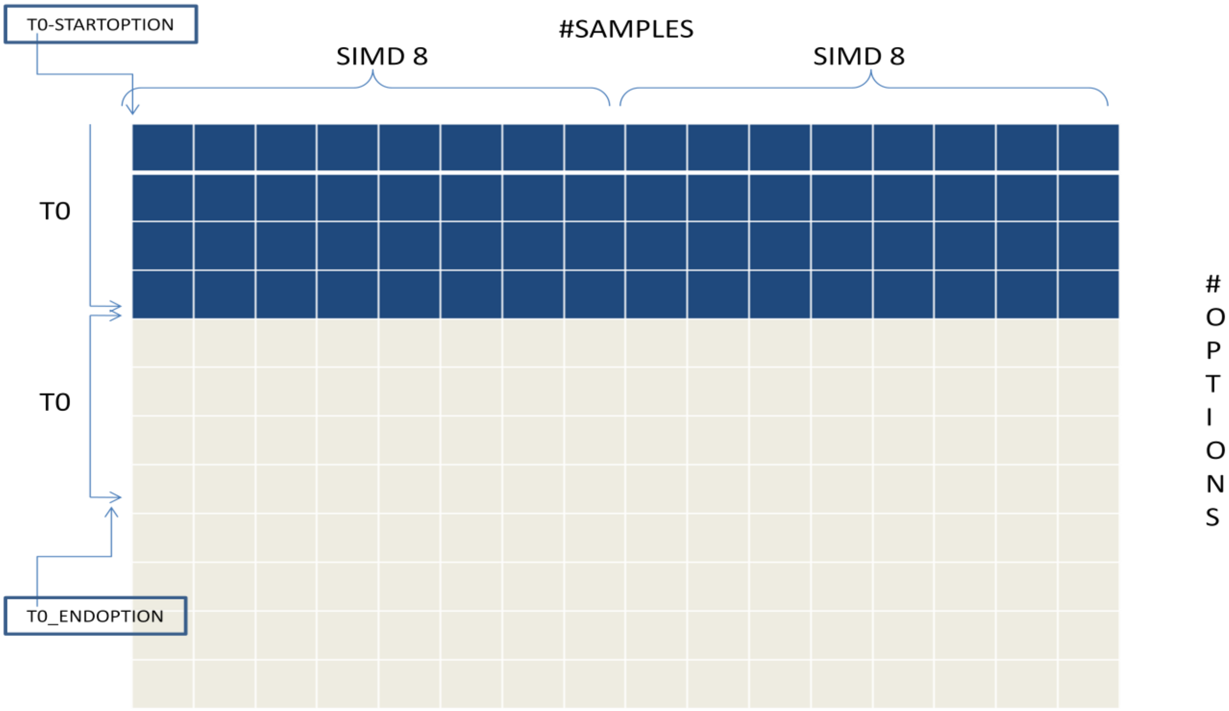


FIG 11. MONTE CARLO GRID 2

4.4. CONCLUSION

The summary of optimizations is shown in the graph below. Implementation and Optimization of Monte Carlo European Option Pricing was completed. This exercise was presented at GPGPU Workload meetings and was well received.

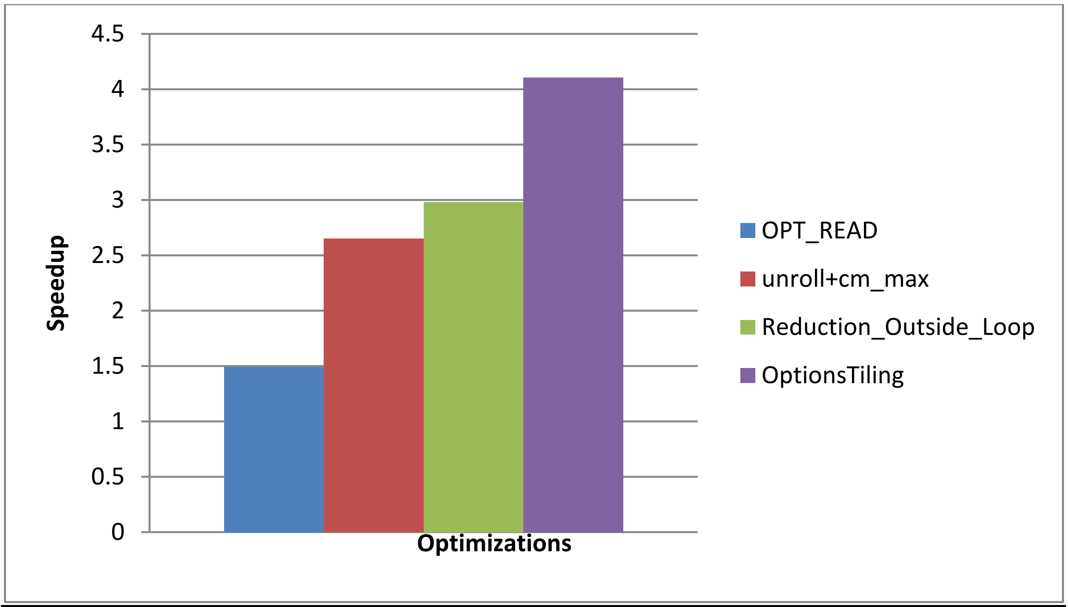


FIG 12. MONTE CARLO OPTIMIZATIONS

5. SILICON MEASUREMENTS FOR VIDEO PLAYBACK

5.1. INTRODUCTION

The goal of this exercise is to identify, understand and analyze the bottlenecks in video playback scenario and analyze the contribution of the components of the media pipeline in the system when media player is running.

5.2. VIDEO PLAYBACK PIPELINE

A general video playback pipeline shown below consists of a demux leading to video decode and an audio decode path later merging into AV Sync (not shown below)[14].

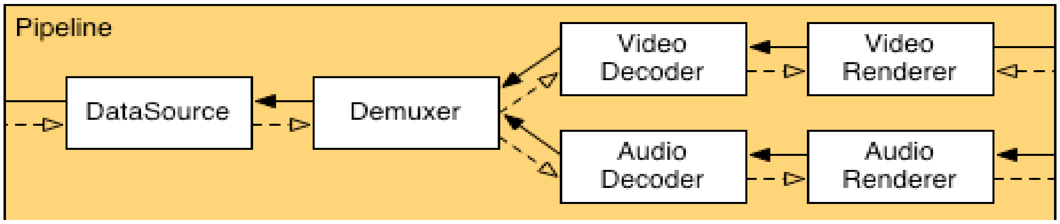


FIG 13. VIDEO PLAYBACK PIPELINE

5.3. SCENARIOS OF VIDEO PLAYBACK

To help with measurement of audio and video overhead, we distinguish 5 scenarios of playback:

1. System Idle: In this scenario, we profile the system with no application running.
2. System running with hardware accelerated Flash player: In this scenario, we profile the system with Flash player application running with hardware acceleration knob enabled. This means that flash player uses Gen(GT or GPU) resources which makes the CPU busy cycles less since the work is being offloaded ie CPU is not very busy.
3. System with Flash player without hardware acceleration: In this scenario, hardware acceleration knob is disabled in Flash player GUI. This means that instead of offloading work off to Gen resources, CPU itself is doing more work ie more number of CPU busy cycles.
4. System with Flash Player with hardware acceleration – audio disabled: In this scenario, we use a utility over the original video which can disable the video and play only audio. Motive of this is to understand the contribution of the audio pipeline stream by itself.
5. System with Flash Player with hardware acceleration – video disabled: In this scenario, we use a utility over the original video which can disable the audio and play only video. Motive of this is to understand the contribution of the video pipeline stream by itself.

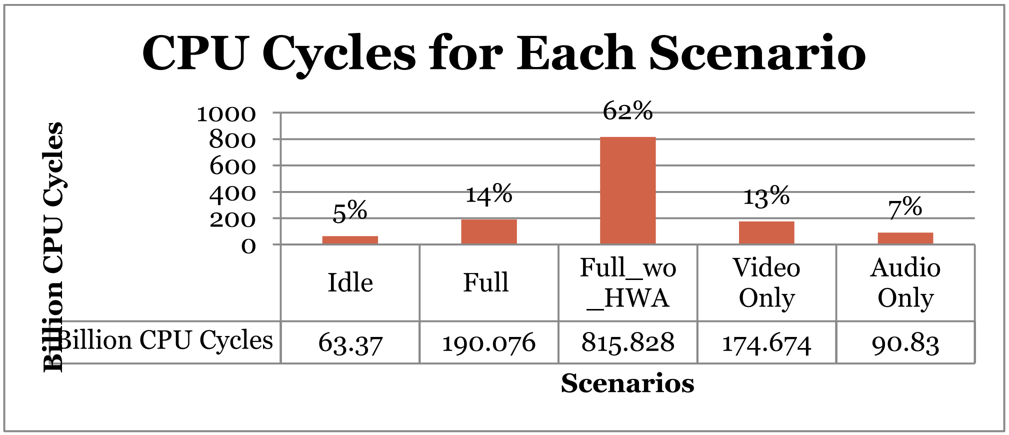
6. Emulating video decode path through Media SDK [15]: Sample code for “decode” available through Media SDK can be modified to emulate the video decode path in the media playback pipeline by disabling the expensive file writes at the end of decode path. The Media SDK however decodes frames in full throttle mode as against the player which is at a particular frame rate (for example, 30 frames per second). Thus, we require implementation of software interrupt in the media SDK decode.

Profiling was done on Intel Vtune Amplifier XE performance profiler and we were interested in targeting the entire system for profile with hardware counter for CPU busy cycles for each scenario. Intel VTune Amplifier XE [16] helps analyze the algorithm choices and identifies where and how an application can benefit from available hardware resources. Intel® VTune™ Amplifier XE enables analysis of the system as a whole as opposed to focusing on a particular application/process. Also, we measured the contribution of each process towards CPU Utilization (represented in the graphs are only the most common processes).

a. MEASUREMENT DETAILS

We used Flash player version 10.2 running the test vector video “Elephants Dream”, a ten minutes video in MP4 format.

b. TOTAL CPU BUSY CYCLES (as a percentage of total CPU cycles)



	Idle	Full	Full_wo _HWA	Video Only	Audio Only
Billion CPU Cycles	63.37	190.076	815.828	174.674	90.83

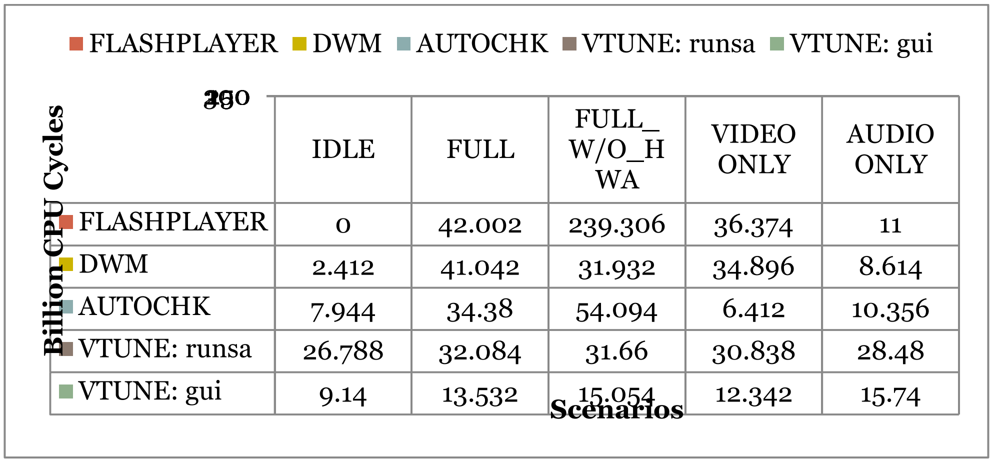
FIG 14. CPU CYCLES FOR EACH SCENARIO

c. PROCESSES WITH HIGH CPU UTILIZATION

- Flash Player
- DWM (OS):Responsible for the graphical effects such as live window previews and a glass-like frame around windows (Aero Glass), without draining CPU

- Autochk (OS): Part of the Microsoft Windows Operating System. It allows functionality to revert core system settings back to their original state, and to backup where necessary
- VTune: We notice that the Vtune profiler itself is a large consumer of CPU cycles. Yet, we avoid using a different profiler since no other profiler gives process/module level details of CPU utilization.

d. CONTRIBUTION OF COMMON PROCESSES



	IDLE	FULL	FULL_ W/O_HWA	VIDEO ONLY	AUDIO ONLY
FLASHPLAYER	0	42.002	239.306	36.374	11
DWM	2.412	41.042	31.932	34.896	8.614
AUTOCHK	7.944	34.38	54.094	6.412	10.356
VTUNE: runsa	26.788	32.084	31.66	30.838	28.48
VTUNE: gui	9.14	13.532	15.054	12.342	15.74

FIG 15. CONTRIBUTION OF COMMON PROCESSES

e. CONCLUSION

Since this study is still in progress and many of our conclusions are yet to be validated, we have not published our conclusions. In the course of this activity, we have learnt several important aspects of the media playback pipeline. Operating System (OS) activity increases with increase in player's activity. OS activity also depends whether player uses GPU HW resources or not. This shows contention of GPU resource across OS and player.

6. CONCLUSION

The end of the internship saw several learnings in the GPGPU and Media Performance Studies front. The first exercise on impact of Shared Local Memory (SLM) on performance of *NBody* kernel involved implementing three additional kernels to test SLM impacts, thus enabling us to make fair observations of impact of SLM. Sandy Bridge (SNB) cards showed that performance is independent of SLM i.e. all implementations perform the same on SNB. On ATI card, for a problem size of less than 2K particles, SLM version performs better and for all other problem sizes, no-SLM version performs better. Comparing ATI and SNB, SNB is 2x slower than ATI for problem size less than 1K and for all other problem sizes, 10x slower than ATI.

Our second exercise was to compare and understand SGEMM kernel implementation in OpenCL and C-for-Media (a language intended to support high level programming of media kernels). The goal of this exercise was to understand how the new language compared against OpenCL in performance. CM kernel performs ~30 x better than OpenCL version for large matrix dimensions.

We gained a deeper insight into C for Media language by implementing a financial kernel-Monte Carlo European Options in C for Media. Working on optimizations over the basic implementation exposed us to many BKMs (best known methods) of optimizing a kernel in CM.

Overall, the internship gave me a broad overview of GPGPUs and also exposed me to several technical aspects of GPGPUs. The study on media performance exposed me to practical learnings on media front. Apart from technical insight, the project gave me confidence to handle parallel time sensitive tasks in an organized manner.

7. REFERENCES

1. N. Arora , A. Shringarpure , R.W. Vuduc, Direct N-body Kernels for Multicore Platforms, *Proceedings of the 2009 International Conference on Parallel Processing,* p.379-387, September 22-25, 2009

2. Portegies Zwart S., McMillan S., Groen D., Gualandris A., Sipior M., Vermin W. *A parallel gravitational N-body kernel (2008) New Astronomy,* 13 (5), pp. 285-29

3. J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80--113, Mar. 2007.

4. A. Munshi, "OpenCL," http://s08.idav.ucdavis.edu/munshi-opencl.pdf, 2008.

5. "OpenCL API 1.0 Quick Reference Card" http://www.khronos.org/files/opencl-quick-reference-card.pdf

6. "Applications for Parallel Computing" http://www.cs.berkeley.edu/~yelick/cs267

7. OpenCL Capturer and Profiler tutorial. (Intel)

8. Gen Execution Units Architecture tutorials. (Intel)

9. C for Media Language Specification. (Intel)

10. C for Media Usage Models (Intel)

11. C for Media API Specification (Intel)

12. Lai, Yongzeng and Jerome Spanier. "Applications of Monte Carlo/Quasi-Monte Carlo Methods in Finance: Option Pricing", http://citeseer.ist.psu.edu/264429.html

13. Craig Kolb, Matt Pharr. "Option pricing on the GPU". GPU Gems 2. (2005) Chapter 45

14. "Intel® Media SDK 2.0: High-Performance Video Encoding, Decoding and Preprocessing" http://software.intel.com/en-us/articles/intel-media-sdk-20-high-performance-video-encoding-decoding-and-preprocessing/

15. Intel® VTune™ Amplifier XE 2011: http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/