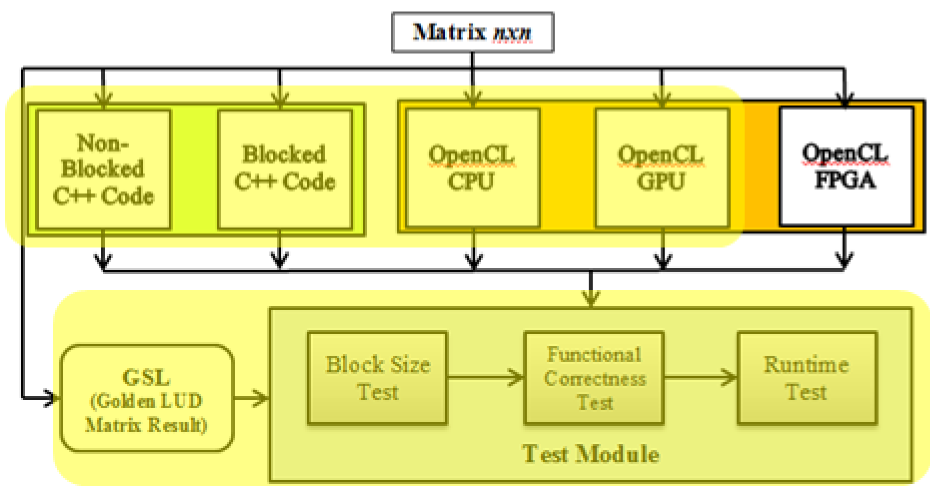
The Edward S. Rogers Sr. Department of Electrical and Computer Engineering **University of Toronto**
ECE496Y Design Project Course **Group Final Report**
Title: Multi-Platform LU-Decomposition Solution in OpenCL

Project I.D.#:	2011027	
Team members: (Select one member to be the main contact. Mark with '*')	Name: Ehsan Nasiri (995935065)*	Email: ehsan.nasiri@utoronto.ca
	Rafat Rashid (996096111)	rafat.rashid@utoronto.ca
	Saurabh Verma (995858237)	saby.verma@utoronto.ca
Supervisor:	Vaughn Betz (vaughn@eecg.toronto.edu)	
Section #:	8	
Administrator:	Cristiana Amza (amza@eecg.toronto.edu)	
Submission Date:	March 22, 2012	

Executive Summary

The purpose of our project was to write a fast OpenCL LU-Decomposition (LUD) solution for the Intel/AMD CPU/GPU and Altera's FPGA and record the amount of recoding required to optimize the algorithm for these platforms. LUD is the mathematical operation which factors a given matrix into the multiplication of a lower triangular and an upper triangular matrix. The complexity of many problems in different fields like biology, circuit design and discrete graphics boils down to this operation. Unfortunately the algorithm has a high computing complexity of $O(n^3)$. Even with today's high-end computing devices, a LUD operation could take hours to days to finish for large matrices. Therefore a cross platform LUD solution will be useful, both for ongoing research in this field and in the industry.

The deliverables are 1) **Blocked** and **Non-Blocked** C++ serial implementation of the algorithm and 2) Blocked **OpenCL** implementations tailored for Multi-core CPU, GPU and **FPGA** (refer to Appendix A for the bolded terms). The project modules are shown in the diagram below.



Project Modules: Highlighted modules have been successfully completed

We successfully met our objective of beating the runtime of the Blocked C++ algorithm run on CPU with our OpenCL CPU/GPU algorithms. We also developed the Test Framework that was used to evaluate and improve our algorithms further. We used the GNU Scientific library (GSL) to ensure our algorithms were producing the correct results. We were not able to optimize the OpenCL algorithm for FPGA due to problems with the DE4 board. We were informed by Altera that the board suffers from a large voltage drop when most of the device resources were utilized.

Table of Contents

Appendices

1. Introduction

This report describes the *Multi-platform LU-Decomposition Solution* project carried out by our team as a part of ECE496 design project course. The report will start by providing the motivation behind the project, followed by discussion of the implementation of our design, testing, and verification of our results. The report concludes by providing our useful findings about high-performance code development, and suggestions for future work.

1.1. Background and Motivation (author E. Nasiri)

The project's goal was to create a high performance LU-Decomposition solution using **OpenCL** that can run on multi-core Intel/AMD CPU, GPU and **FPGA**. **LU-Decomposition** (LUD) is the mathematical operation which factors a matrix into the multiplication of a lower and an upper triangular matrix (refer to the glossary in Appendix A for **bolded** terms). This operation is the very basic method for finding the inverse of a matrix as well as solving a system of n linear equations and n unknowns [1]. In fact, this operation is so fundamental that the complexity of many problems in different fields boils down to the complexity of this single operation. Consider the following examples from four different fields where LUD is used: Electronic Circuits, Computer Networks, Chemical Engineering, and Economics.

In the software simulation field, programs such as **SPICE** (Simulation Program with Integrated Circuit Emphasis [2]) create a system of equations using known currents and voltages for every node of an analog circuit and solve for all unknown parameters to simulate [3]. Similarly, in the Computer Networks field, knowing that the number of data packets entering and exiting a node in a network is always equal, a system of equations is formed to solve the Traffic Network Matrix [4]. When studying chemical reactions, a set of equations is formed by balancing the reactant and resultant chemicals' molarities and a Stoichiometric Matrix is formed [5]. Solving results in finding the unknown weight/volume of the chemicals. Lastly, we can see the use of solving a system of linear equations in Wassily Leontief's input-output model for which he won a Noble Prize in Economics. In Leontief's closed model, an economic system is considered closed if it satisfies its own needs. We can represent the economy as *n* independent industries, each of which have a level of consumption from other industries and have a total production output. Using this idea, an input-output matrix is formed and solving the system of equations using this matrix provides us with the production level of each industry in order to satisfy the internal and external demands [6].

As we can observe, the applications of this basic algebraic operation is extensive. Further, the complexity of these applications is the same as solving a LUD problem. Unfortunately, this operation has a high cost (high computing complexity). In algorithmic terms, it has a complexity of $O(n^3)$ which means the run-time increases in a cubic manner as the size of the matrix increases [1]. Even with high-end computing devices, the operation could take hours or days to finish for a matrix of considerable size (e.g. a 20,000 by 20,000 matrix).

The LUD problem has been around for decades and mathematicians and computer scientists have been trying to optimize the algorithm for a lower runtime. Currently, there are existing solutions that are used in both industrial and academic fields. Some are implemented in C/C++ such as within the GNU Scientific Library (GSL) [7]. This library is open-source and freely available. Other solutions are lower level like the Intel's Math Kernel Library (MKL). MKL contains LUD functionality but it is specifically optimized to be run on Intel CPUs and it costs $399 for a single-user license [8]. Lastly, AMD has recently developed a LUD algorithm within their latest Development Kit for AMD GPUs [9]. These existing solutions are specifically designed to target a specific platform (such as a CPU or GPU).

In recent years, the Khronos group consisting of companies such as Intel, AMD, Nvidia, Texas Instruments and Altera has been developing a cross-platform standard programming language for parallel development known as **OpenCL**. In this project, we took advantage of this language to develop a LUD solution that runs on multiple computing platforms.

1.2. Project Goal and Requirements (author R. Rashid)

The goal of our project is to create a high performance multi-platform LUD solution using OpenCL that can run on the following platforms: Intel/AMD CPU, GPU and Altera's FPGA. The subsections below define the requirements used to evaluate the success of the project.

1.2.1. Functional Requirements

The project required a working version of the components defined below. A component is working if it 1) compiles and runs on the target platform(s) and 2) produces the correct L and U output matrices. Refer to Appendix D for Validation and Acceptance Tests.

<table>
<tr><th></th><th>Component</th><th>Target Platform(s)</th></tr>
<tr><td>1</td><td>Serial Non-Blocked LU Decomp Algorithm in C</td><td rowspan="2">• OS: Windows 7
• Will run on Intel/AMD CPU</td></tr>
<tr><td>2</td><td>Serial Blocked LU Decomp Algorithm in C</td></tr>
<tr><td>3</td><td>Serial Blocked LU Decomp Algorithm in OpenCL</td><td rowspan="2">• OS: Windows 7
• Intel/AMD CPU</td></tr>
<tr><td>4</td><td>Parallelize the Blocked LU Decomp Algorithm for optimal performance</td></tr>
<tr><td>5</td><td>Modify and optimize the Blocked LU Decomp Algorithm to run on the GPU</td><td rowspan="2">• OS: Windows 7
• Intel/AMD discrete GPU
• Altera DE4 Board - FPGA</td></tr>
<tr><td>6</td><td>and FPGA</td></tr>
</table>

1.2.2. Constraints

The list below defines the constraints imposed on the project:

- OpenCL SDK 1.1:

  The OpenCL code will not be backward compatible with earlier versions of the SDK. Compatibility with future SDK versions will not be guaranteed.
- Intel/AMD’s OpenCL compiler for their CPUs/GPUs
- Altera’s OpenCL compiler for their FPGAs
- Single-precision floating point operation on **dense** input matrices

1.2.3. Objectives

The list below defines desirable goals of the project:

- Optimize the blocked OpenCL code to minimize runtime on the target platforms.
- Minimize the amount of recoding required to move from one platform to another. Ideally have only one working version of the algorithm that can compile and run on all of the targeted platforms.
- Record how much recoding is required for multiple platform compatibility.
- Implementation should be parameterized in a way that it can be easily ported to new generations of the target platforms.

2. Final Technical Design

In this section we introduce the modules of the project and provide some explanation about each component. In the following sections (Section 3, 4, and 5), we provide details of our algorithm design in C++, OpenCL for CPU, and OpenCL for GPU, respectively.

2.1. System Level Overview (author: S. Verma, R. Rashid)

The project was separated into three modules (as shown in the system block diagram below):

1. C++ LU-Decomposition development
2. OpenCL code development for CPU, GPU and FPGA
3. Testing, Verification and Results

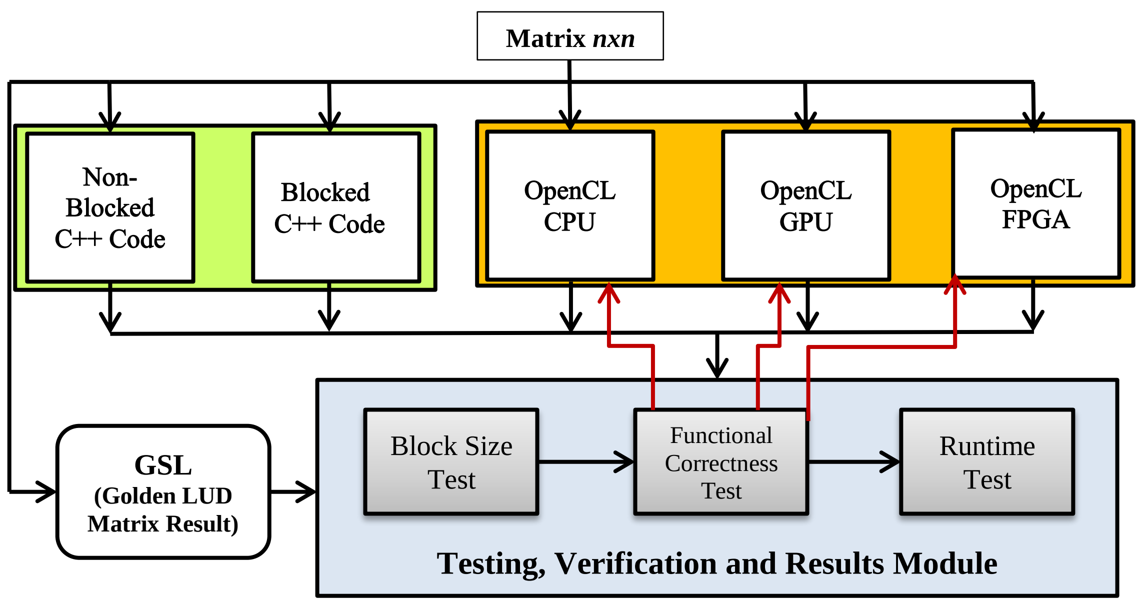


Figure 1: The System Block Diagram

The first stage (shown in green in the above diagram) was to research the mathematical and programming advancements in the field (literature review) and to use this knowledge to implement high-performance LUD Blocked and Non-Blocked algorithms in C++. We used the "Portable and Scalable FPGA-Based Acceleration of a Direct Linear System Solver" academic paper [10] as one of our guides for designing these algorithms. In this paper, Wei Zhang implements the LUD algorithm directly in hardware using an FPGA. This stage was completed over the summer of 2011.

The second stage (shown in orange) was the development of a multi-platform OpenCL algorithm capable of running on CPU, GPU and FPGA. Here we took advantage of the OpenCL library's ability to execute tasks in parallel and the capabilities of the target platform. Lastly, the third stage (shown in blue) was to develop an automated testing platform that was utilized to regularly check the correctness of our algorithms and provide us with feedback on their performance compared to the C++ code that was developed in Stage 1. This iterative feedback (the red arrows) helped us improve the run-time over the course of the project.

As shown in Figure 1, the only input to our algorithm is a n x n matrix on which LUD operation needs to be performed. The input matrix is fed into the modules that implement the algorithm for the respective platforms in C++ and OpenCL. The computed result is then fed into the Test Module for verification and runtime performance comparisons.

2.2. Module Level Descriptions and Design (author: S. Verma)

This section provides a more detailed description of the 3 modules described above.

2.2.1. C++ Development Module

This module involved the development of two versions of the LUD algorithm using the C++ programming language: **Blocked** and **Non-Blocked**. Both produce an in-place matrix that consists of the Lower (L) and Upper (U) triangular matrices. We used Wei Zhang's paper [10] to design both algorithms. Developing this module first allowed us to:

1) Become familiar with the 2 variation of the algorithm in a familiar environment
2) Quantitatively justify why we chose to develop the Blocked version in OpenCL
3) Provide a benchmark for comparing runtime performance of our OpenCL algorithms

Non-Blocked LU-Decomposition Algorithm

Also called Simple LU Factorization, this algorithm brings the entire matrix into memory prior to performing the LUD computation in-place. The pseudo-code and detailed description of the algorithm can be found in Appendix E.

Blocked LU-Decomposition Algorithm

There are three common variants of the block LU Decomposition [10]; and we implemented the "right-looking" version. In this method, the matrix is divided into blocks of which there are 4 types. As shown in Figure 2(a) below, the block operated on (black) depends on the top-most and left-most blocks (gray). The computation proceeds as shown in Figure 2(b). The pseudo-code for this algorithm can be found in Appendix F.

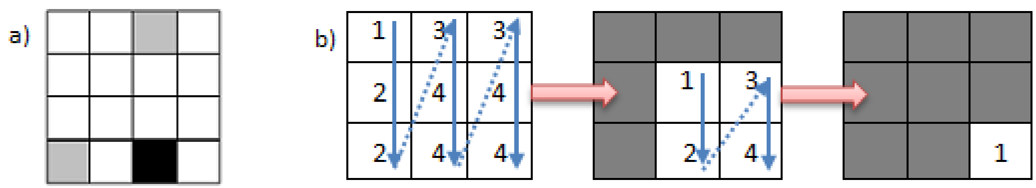


Figure 2: a) Block dependency b) Blocked Algorithm computation

2.2.2. OpenCL Code Development Module

This module comprised the bulk of our project. It consisted of 3 phases: OpenCL LUD implementations for CPU, GPU and FPGA. We started with the CPU implementation in October 2011. By December, we started concurrently working on the GPU by porting what we had for the CPU at the time. Both efforts were successful. The work on FPGA was abandoned due to problems with the FPGA DE4 board. We were informed by Altera that the board suffers from a large voltage drop when most of the device resources were utilized.

2.3. Testing, Verification and Results Module (author: S. Verma)

The four components of the testing infrastructure are written in Python and described below:

2.3.1. Input/Golden Output Matrix Generator

The matrix generator module generates 2-D matrices that are used as an input for both C++ and OpenCL LU-Decomposition algorithms. The module generates two types of matrices:

Input Matrices: The algorithm generates matrices of sizes varying from 1,000 to 10,250. The diagonal elements of these matrices are greater than the sum of all the other elements in the same row. This ensures the LUD operation converges.

Golden Matrices: We used GSL's [7] *gsl_linalg_LU_decomp* function to generate "golden" result matrices which were used to check the correctness of all our algorithms.

2.3.2. Functional Correctness Test

The algorithm verifies the correctness of the L&U matrix produced by our LUD algorithms by comparing it with the golden matrix produced by GSL, against various input matrices. The comparison (% error) is calculated using the following formula:

$$\%\ \text{error} = \frac{diff}{golden\ matrix\ magnitude} = \frac{\sqrt{\sum_{i=0}^{matrix\ size}(alg\_result[i] - golden\_result[i])\text{^}2}}{\sqrt{\sum_{i=0}^{golden\ matrix\ size} golden\_result[i]\text{^}2}}\ \text{x}\ 100$$

The numerator is an accumulator for the degree of error we observe between our algorithm's output and that of GSL. This error grows proportionally to the size of the matrix. Therefore, we normalize this error by dividing it by the magnitude of the golden matrix. The threshold for error is 0.001. If the error is greater, our algorithm is considered incorrect. A sample output of the functional correctness check algorithm is shown in Figure 3.

```
Results:
Test Case: 100x100.txt: Error = %7.48328430447e-05
Test Case: 12x12.txt: Error = %7.5541057351e-05
Test Case: 3x3.txt: Error = %0.000445755111815
Test Case: 2000x2000.txt: Error = %9.67328412777e-03
Test Case: 60x60.txt: Error = %0.0000163423244508
Test Case: 6x6.txt: Error = %2.24132240428e-08

All Test Cases ..... Passed!

The algorithm is performing correctly.
```

Figure 3: Sample Output of the Functional Correctness Test

2.3.3. Optimal Block Size Test

The OpenCL Blocked algorithm's performance is highly dependent on the block size used. This test runs a 10k x 10k input matrix with different block sizes to determine the optimal block size that results in the lowest runtime on the target CPU and GPU platforms.

2.3.4. Runtime Measurement Test

The runtime comparison test plots the runtime of two or more LUD algorithms against matrix size. The block size used is determined from the Block Size test discussed above. This test was used to compare the runtime of the C++ Blocked algorithm with the OpenCL algorithms.

All OpenCL algorithms developed went through all stages of the testing infrastructure, as illustrated in Figure 4. The feedback from this iterative process helped us in optimizing our algorithms for different platforms and make sense of the results.

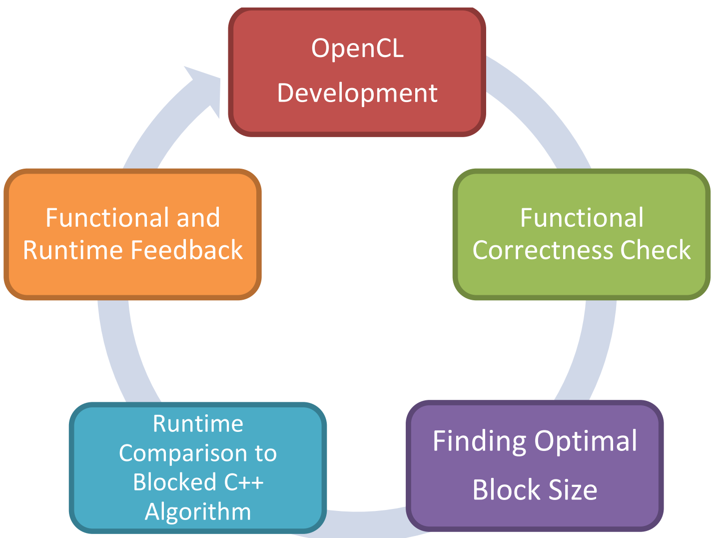


Figure 4: OpenCL Algorithm Design Process

2.4. Assessment of Final Design (author: S. Verma)

For the Non-Blocked LUD algorithm, all matrix elements must be accessible during the computation. For a large matrix size such as 10,000, it requires at least 10,000x10,000 single-precision numbers (roughly 0.4 GBytes) to be in memory. This is far too large to be stored in the processor's cache, and therefore must be stored in global memory. Fetching data back and forth from the global memory is more time consuming than fetching it from processor's cache. The Blocked algorithm operates on at most three blocks at any given time. If the block sizes were chosen correctly, they could fit into the L1, L2 or L3 caches of the processor, thus reducing the time required to fetch the data. Since the project goal was to optimize runtime for very large matrices, the team decided to implement all OpenCL algorithms using the blocked approach as it performs better due to more efficient utilization of caches.

2.5. CPU and GPU Architecture Considerations (author: S. Verma)

For our OpenCL development and benchmarking we used the machine named **Glados** provided to us by our supervisor. Figure 5 shows the machine's hardware specification:

CPU	GPU	FPGA	OpenCL
Intel Core i7-2600K @ 3.4 GHz	AMD Radeon HD 6900 Series 2 GB Memory 880 MHz core clock	Stratix IV GX 531,200 Logic Elements 27,376 Kbits on-chip memory	Version 1.1
RAM: 16 GB OS: Windows 7 (Professional Edition) Microsoft Visual Studio 2010 (Ultimate Edition)			

Figure 5: Target Machine "Glados" Hardware Specification

Understanding the architecture of both the CPU and GPU was important as it provided insight into how much parallelism can be inherited within each device to achieve full performance. Having knowledge of the size of caches and work groups helped us find the optimal block size for each device. Appendix G provides the specification of Glados' CPU and GPU. Specific considerations are discussed in Section 4 for CPU and Section 5 for GPU. More information can also be found in Appendices H-J.

3. C++ Module Development Details (author: S. Verma)

Initially, our C++ Non-Blocked algorithm was beating the runtime of our C++ Blocked algorithm on a CPU as shown in Figure 6 below.

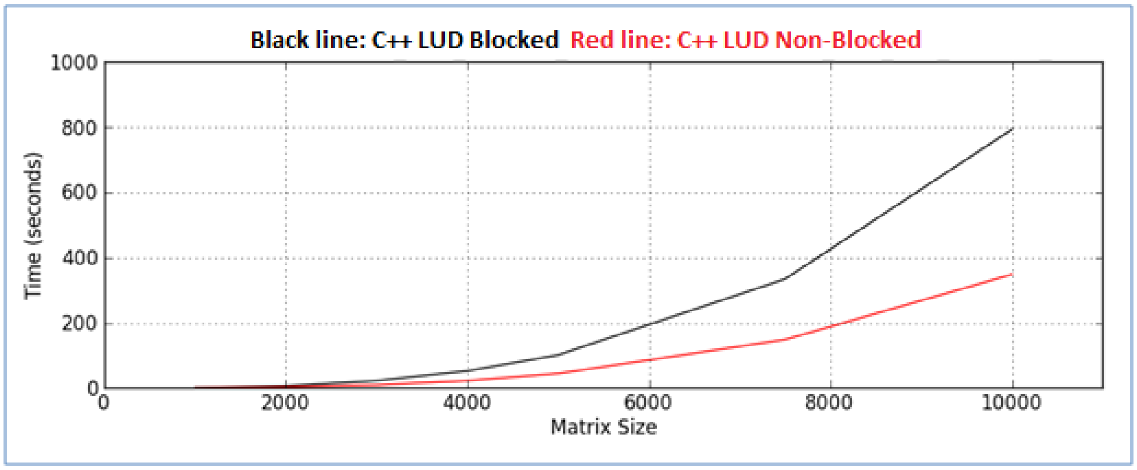


Figure 6: Runtime performance of Blocked and Non-Blocked algorithms

This result was conflicting with our initial thinking that the Blocked approach should decrease runtime for large matrices. After a lot of testing, our supervisor found that we are passing the matrix and block size parameters **by reference** to many sub-functions even though we were not modifying these parameters within them. He indicated that the Intel CPU would not store these variables into registers or caches and would dereference it every time it wants to use the value, thus making the algorithm run much slower. From this feedback, we modified the blocked algorithm and improved runtime by a factor of three. The new algorithm performed much better than the Non-Blocked algorithm as shown in Figure 7, as was expected.

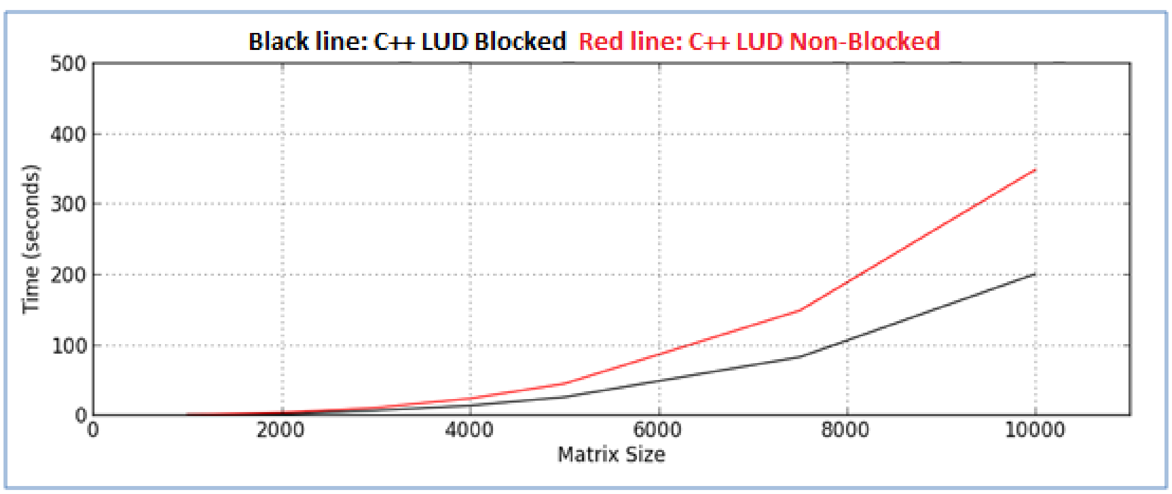


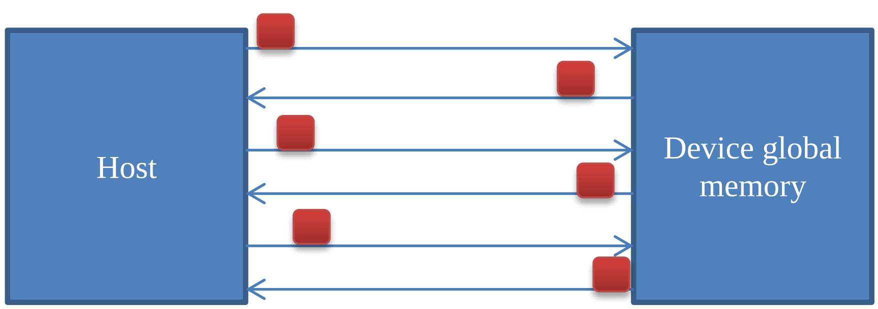
Figure 7: Runtime performance of updated Blocked and Non-Blocked algorithms

4. OpenCL CPU Algorithm Development Details **(author: E. Nasiri)**

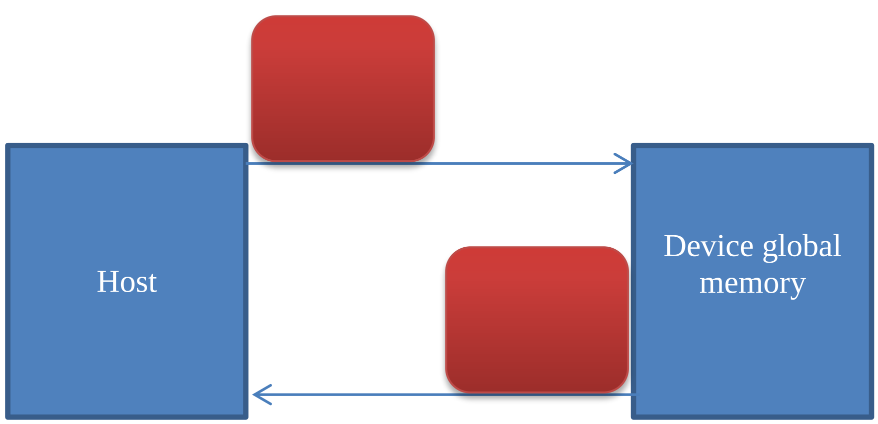
Due to architectural differences between a CPU and a GPU, such as memory hierarchy, one should have the target platform in mind when developing an OpenCL algorithm. Therefore, our final design includes an algorithm that is specifically optimized to run on CPUs. Our final OpenCL algorithm for CPU has gone through several stages of optimization and is able to beat the performance of our C++ Blocked LUD code. These optimizations are described below.

4.1. Performing few data transfers between the host and the compute device

Transferring data between the host and the compute device is an expensive operation. In OpenCL, the user is able to copy data from the host to the device using `clCreateBuffer` and after the compute device has performed operations on the data, the host can read back the results using `clEnqueueReadBuffer`. These two operations are expensive and therefore we use each of them once. We create a buffer to hold the entire matrix, perform the entire LU-Decomposition operations and then read back the buffer once in the end.



Transfer small blocks back and forth to device global



Transfer the entire matrix once to device global memory

Figure 8: Transferring Submatrix vs. Matrix to the compute device

4.2. Pushing work from host to compute device

It is easy for a beginner in OpenCL to fall into the trap of keeping some of the algorithmic logic on the host side. Consider the following part which is a sample from our algorithm:

Host.c

```
for(col=0;col<n; col++)
{
    //launch kernel
   clEnqueueNDRangeKernel(…, col, …)
}
```

Host.c

```
//launch kernel

clEnqueueNDRangeKernel(…, n, …)
```

Kernel.cl

```
__kernel void mykernel(const int col, …)
{
    //get row using thread id
    row = get_global_id(0);

    //col was passed to kernel by host

    //use row and col to do something
}
```

Kernel.cl

```
__kernel void mykernel(const int n, …)
{
    //get row using thread id
    row = get_global_id(0);

    for(int col=0, col<n; col++)
   {
     //use row & col to do something
   }
}
```

Figure 9: Pushing loops from the host to the kernels

The two algorithms shown will do the same work. However, the algorithm on the right runs faster. The algorithm on the left 'enqueues' many kernels in a loop. There is an overhead with launching kernels. For example, the host has to set the arguments for each kernel that it wants to launch (using `clSetKernelArg`). The command queue for the target device will also get much larger, and the host has to keep track of when each launched work-group has finished its job and can do something new. This causes the memory usage of the program to increase dramatically. Furthermore, the algorithm on the right allows the OpenCL compiler to see our whole algorithm and perhaps perform some optimization or vectorization.

4.3. Using multiple threads ("work-items") and utilizing the CPU cache

OpenCL provides an excellent infrastructure for the coder to explicitly declare parallelism. Therefore it is important to verify what can be done in parallel in the algorithm. Since we are implementing the blocked LU-Decomposition algorithm in OpenCL, we can have as many threads as the block size (block size is the number of rows or columns in a block). This will allow us to perform many independent arithmetic operations simultaneously. Consider the following case which occurs for processing of blocks of type 1 and 2:

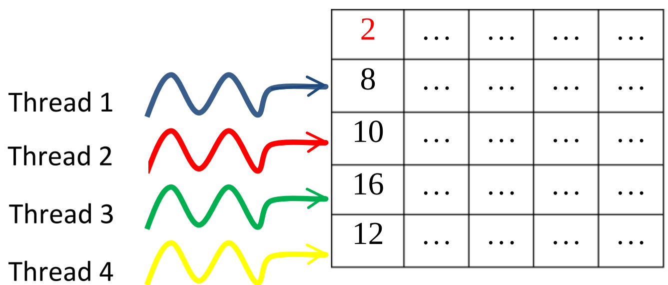


Figure 10: Normalization of a column using multiple threads

In block type 1, we need to normalize every element in the column below the diagonal element by the diagonal element. In the figure above, a block of size 5 is shown and 4 divisions are needed (divide 8,10,16,12 by 2). Using multiple threads we are able to perform all of these independent operations simultaneously. Similarly, when calculating the changes to a row in all block types, we can use as many threads as the block size. The following figure shows how we can achieve this:

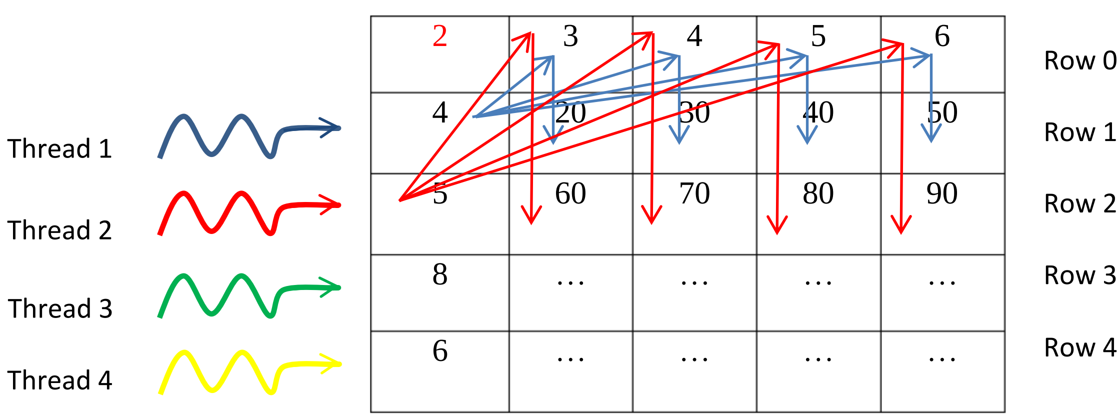


Figure 11: Reducing rows with multiple threads

We need to update the entries on all rows. Each entry will be subtracted by the multiplication of the left-most element on its row (which has been already normalized) by the top-most row by which we're trying to reduce. So, for instance, 20 will become 20 – (4×3) = 8; and 30 will become 30 – (4×4) = 14. As you see, these operations for different rows are independent. So, Thread 1 can update all the entries on row 1, Thread 2 updates the entries on row 2, and so on.

One could also design the algorithm such that thread 1 updates the entries on column 1, and so on. However, the best way to make sure we can utilize the CPU cache is to follow the access pattern as shown in the figure above. Note that Thread 1 is updating adjacent row entries (e.g. 20,30,40, and 50). Once this thread tries to access the first element of the row, the CPU will not only get that element from the memory but also cache a big chunk of data next to it. Since adjacent elements in a row are stored sequentially in memory, it works out best for us to have the access pattern such that each thread accesses one row.

As you can observe, the number threads we launch is equal to the number of rows in a block ("block size"). Here we can see an interesting tradeoff. As we increase the block size, we are in effect increasing the number of threads and therefore introducing more parallelism. On the other hand, as the block size gets bigger (say a block of size 1000), the entire block may not fit in the cache and therefore reduce efficiency.

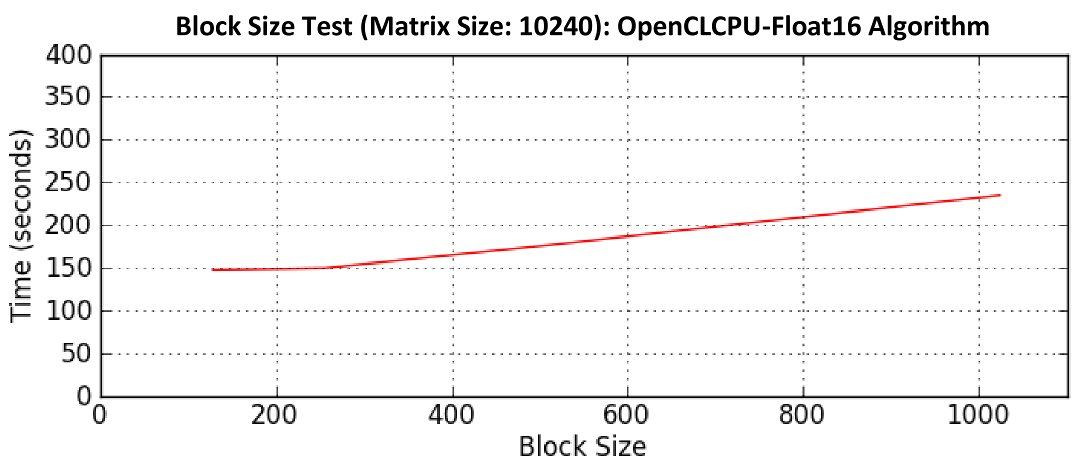


Figure 12: Run-time of OpenCL algorithm as a function of block size

We have found a block size of 250 to be optimum for our given device. Ideally, we would like to be able to fit 3 blocks in the cache (refer to Blocked algorithm description in Section 2.2.1).

Memory usage = 3 blocks × *(250×250 elements)* × *(4 bytes/element) = 750,000 Bytes.*

Since our CPU has 4×32KB of L2 data cache (refer to Appendix G), using a block size of 250 allows us to store three blocks in L2 cache and have faster memory accesses.

4.4. Using SIMD (Single Instruction, Multiple Data)

Today, computing devices that have multiple processing elements are able to perform the same operation, say multiplication, on two sets of data simultaneously. In order to be able to tell the processor to use SIMD, one must guide the compiler to generate such machine level instructions (known as SSE instructions on Intel [12]). In OpenCL, one can use the built-in type known as "*float16*". As the name suggests, float16 stores 16 floating point numbers and when addition, subtraction, or multiplication commands are issued between two float16 containers, the compiler will generate SSE instructions to do the operation on all 16 elements in parallel. The processor may support up to 4-way (or 8-way SIMD) meaning that in order to operate on 16, it will do 4 operations simultaneously, and go to the next 4 until all 16 are calculated. In order to implement this within our algorithm, we add a restriction on the matrix size to be a multiple of 16, and we choose a block size which is also a multiple of 16. Note that this is not a problematic restriction, because for a given matrix of an arbitrary size, we can pad the matrix rows and columns with zeros to increase its size to the closest multiple of 16. Since we now have a block size that is a multiple of 16, we think of each row as having multiple float16 variables on it:

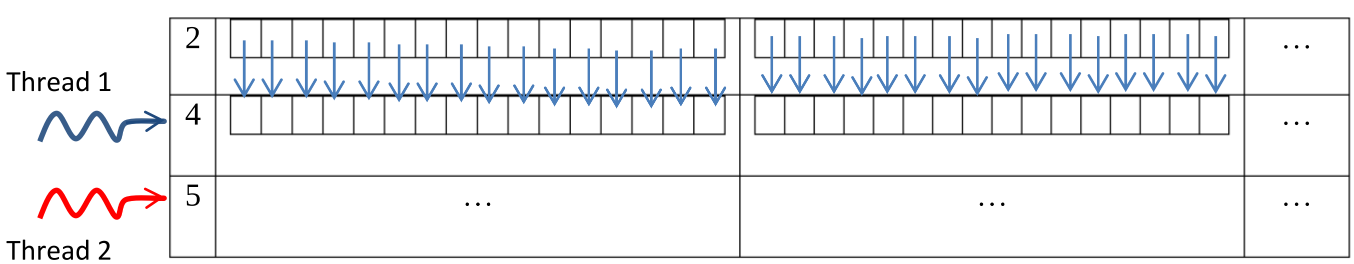


Figure 13: Using multiple Float16 variables on each row

The following graph shows the run-time of two of our OpenCL algorithms, compared to the blocked C code run on CPU.

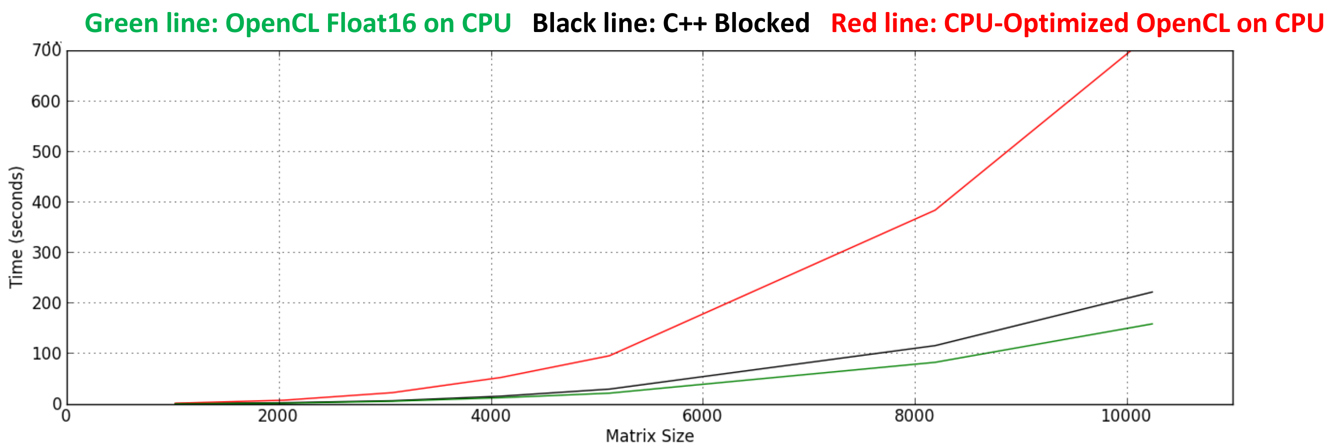


Figure 14: Comparison of 3 algorithm run times (Block size = 256)

The green curve at the bottom is our fastest implementation of the OpenCL algorithm for a CPU which is able to solve a 10,240×10,240 matrix in 135 seconds. This is faster than the blocked C code implementation (black curve). Also note that the effect of using float16 can be seen by comparing the red and green curves. The red curve does not use float16 and it appears that float16 has caused a 4X speedup. This makes sense assuming that the algorithm has resulted in machine code that utilizes the 4-way SIMD processor registers.

Our final algorithm for the CPU uses all the above optimizations. It takes advantage of float16 type and uses parallelism when computing a block (multiple threads work on a block), but different blocks are processed sequentially.

Refer to Section 6 (Project Summary) for Gigaflop calculation of our algorithms.

5. OpenCL GPU Algorithm Development Details (author R. Rashid)

The GPU device is quite different from the CPU (refer to Appendix G for their specs). Recall that on our CPU we can execute up to 1024 work items (threads) in parallel at any given time. Also recall our LUD algorithm generates the same number of concurrent work-items as the block size we use. In our OpenCL CPU implementation, we decided to 1) perform the computations within a block in parallel and 2) the blocks in serial. This is illustrated in Figure 15, with the numbers representing the block types and the blue arrows representing the serial execution of the algorithm from one block to another. As previously explained, we attempted to use a large enough block size to keep the CPU as busy as possible. However we were restricted by the overhead of thread context switching, CPU L1 and L2 cache sizes and memory bandwidth.

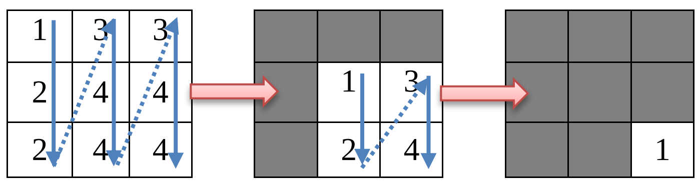


Figure 15: OpenCL CPU Execution of the LUD Algorithm

5.1. GPU Architecture and Design Considerations

Our GPU is organized into 24 **Streaming Multiprocessors** (SMs), each with their own local memory, pipelined processing units and floating point compute units. Each SM can execute up to 256 work items (compared to 1024 on the CPU). This enforced a much lower limit on our maximum block size. The Block Size test was used to determine the size that resulted in the best runtime. The GPU algorithm design followed the iterative process used in designing the CPU algorithm. The Testing and Verification module provided us with performance feedback that we used to further optimize the algorithm. This is illustrated in Figure 2.

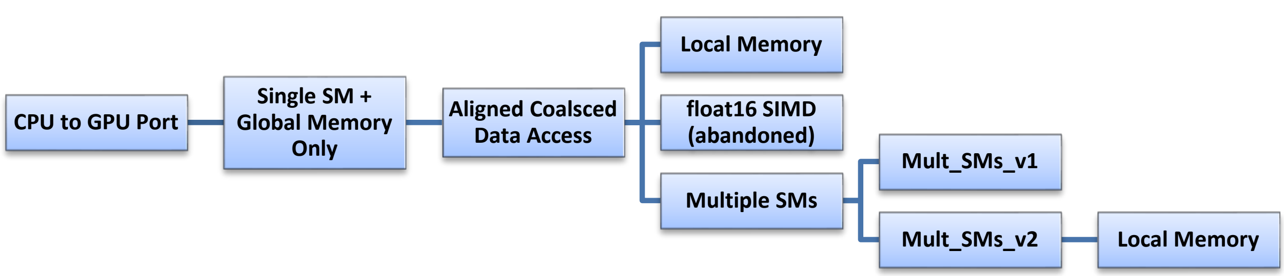


Figure 16: OpenCL GPU Algorithm Iterative Design Process

Initially we ported the CPU code to execute on a single SM on the GPU. The execution of the algorithm remained as illustrated in Figure 15. We observed poor runtime performance compared to the CPU implementation, even with coalesced access and local memory. This was attributed to the other 23 SMs of the GPU being idle. At the final stages of our project, we rewrote the LUD GPU algorithm to utilize multiple SMs of which there are two variants:

1. Multiple **workgroups** computing different parts of a single block (Mult_SMs_v1)
2. Multiple **workgroups** computing different blocks (Mult_SMs_v2)

The design considerations of each of the steps outlined in Figure 16 is discussed below.

5.2. Porting the LUD Algorithm from the CPU to GPU

This step was completed in two iterations. First the OpenCL setup code was modified to target our AMD GPU. The second iteration involved combining the two platform's setup code and adding the ability to dynamically switch between the two devices. This was done to meet our objective of reducing the amount of recoding required to move from one device to another. Prior iterations of the algorithm were also updated to use this "CPU/GPU Switching" code. Having the same setup code across all versions enabled us to make accurate performance comparisons between them. Our first GPU algorithm executed as shown in Figure 15 on a single SM.

5.3. Aligned Coalesced Data Access

The OpenCL CPU algorithm accesses data in a sequential fashion, with each thread accessing a row. We found a coalesced access pattern resulted in better performance on the GPU (but worse performance for CPU), as shown in Figure 17.

Table 1: Data points for Figure 17

Matrix Size	C++ Blocked	Coalesced Kernels on GPU	CPU Kernels on GPU
1000	1	1	1
2000	2	10	14
3000	6	37	45
4000	14	71	112
5000	27	143	219
7500	88	474	744
10000	211	1123	1797

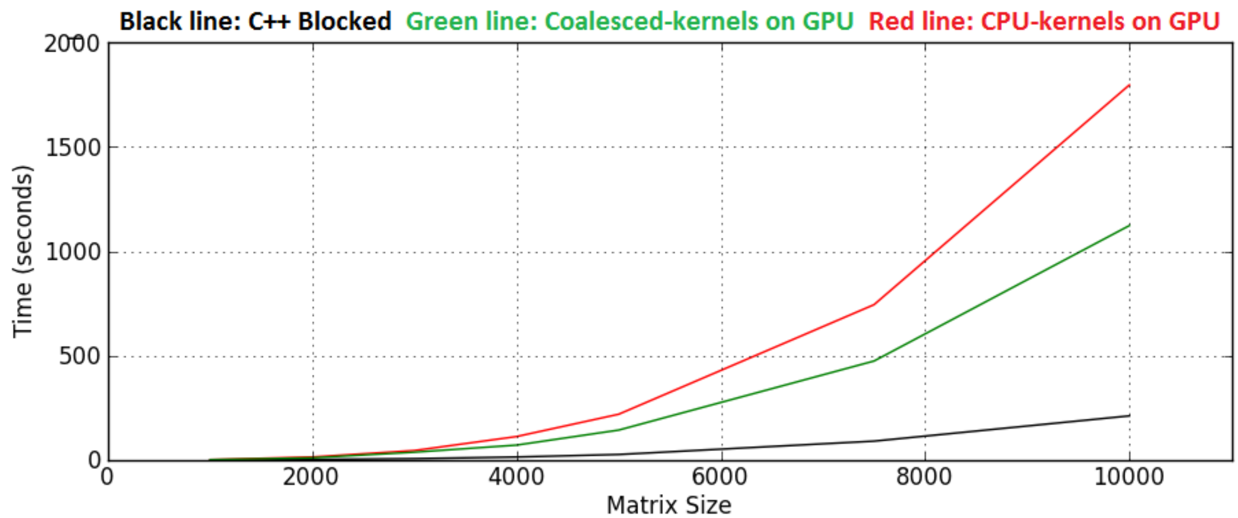


Figure 17: Coalesced access pattern performs better than having each thread access a row of a block

Coalesced read/write corresponds to each thread accessing shared data column wise instead of sequentially (or row wise). Refer to Appendix H to see how this works. This also removed bank conflicts which would otherwise serialize the memory accesses of adjacent threads on the GPU (refer to Appendix J).

We used our Functional Test to verify that the algorithm produced the correct results and the Block Size Test to find the appropriate block size that gave us the best runtime performance. From Figure 18, we observe that block size to be 250. This made sense as the block fits within the 512 kB L2 cache and the data being accessed resided in the global memory of the device.

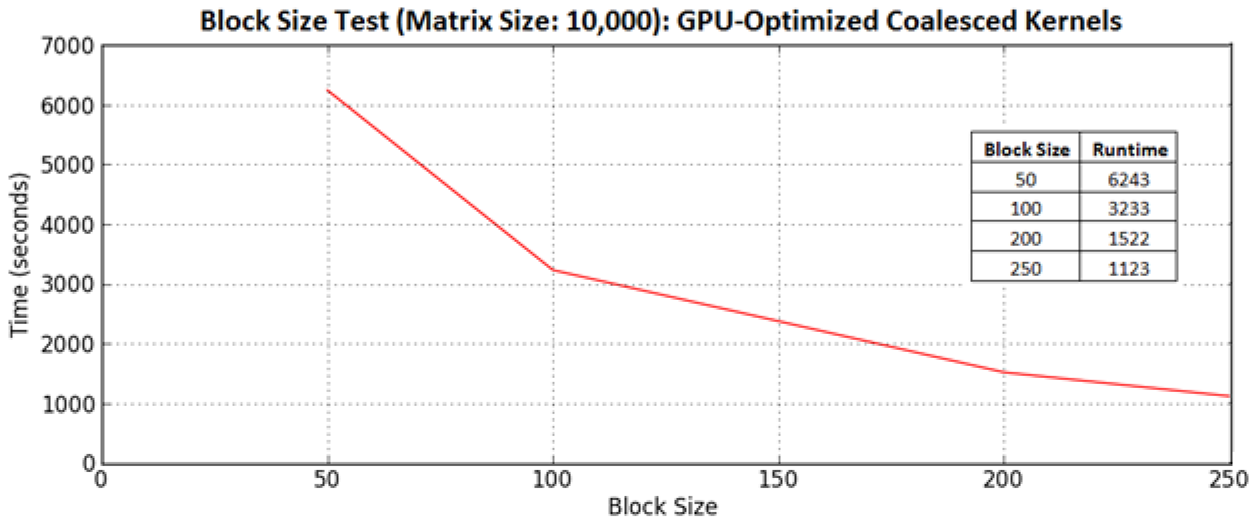


Figure 18: Block Size Test run on GPU algorithm with coalesced read and global memory

5.4. Utilizing Local Memory

Unlike global memory, local memory is private to each workgroup of threads, much smaller in size and faster to perform computations on. The OpenCL algorithm for the GPU was modified to take advantage of this. Prior to computing a block, the block and any dependent blocks (up to 3) were brought into local memory from global memory where the entire matrix was stored. This added the overhead of copying data needed from and to global and local memory. Refer to Appendix I for details on GPU memory layout. With 32kb of local memory, we were able to use a block size of 50 as the storage requirement was:
30kb ≈ (3 blocks) × (50×50) × (4 bytes/element).

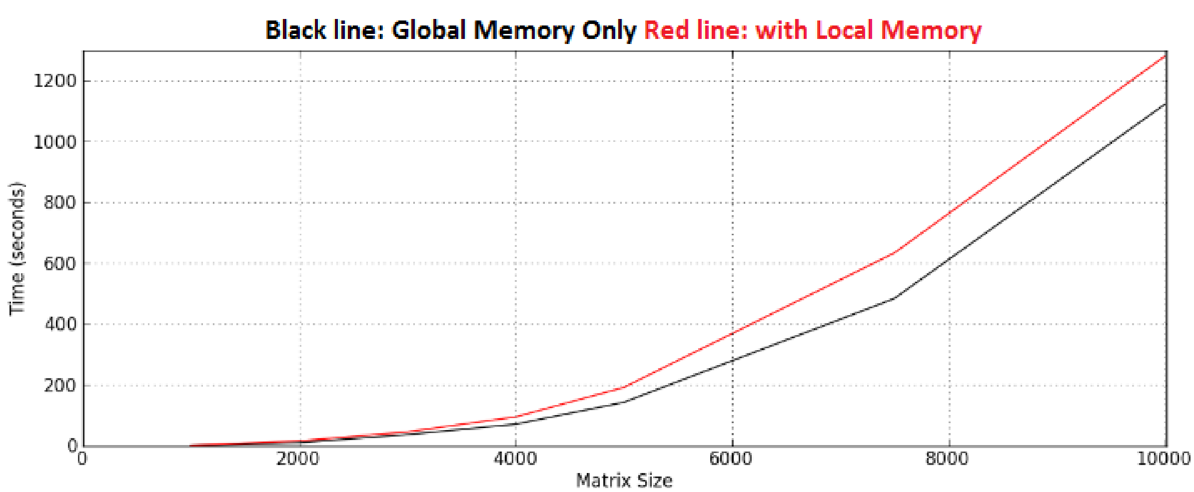


Figure 19: The Coalesced GPU LUD algorithm performs worse with local memory

In Figure 19, the black line was run with a block size of 250 and the red line with a size of 50. By lowering the block size, we reduced the number of concurrent threads used to compute a block from 250 to 50. At the same time, we increased the total number of blocks to be computed by a factor of 5. Note that we are still only utilizing a single SM at any given time.

5.5. Utilizing Multiple Streaming Multiprocessors – Two Variants

The previous versions of the GPU algorithm performed poorly due to the underutilization of GPU resources. The next logical step was to modify the algorithm such that all 24 SMs could be utilized in parallel to perform the LUD computation. We generated two versions. The first uses multiple workgroups to compute different parts of a single block (Mult_SMs_v1) in parallel, with the matrix being computed serially as shown in Figure 15. The second version (Mult_SMs _v2) performs the computation as shown in Figure 20. Each workgroup (which can be routed to a different SM by the GPU) computes a different block in parallel. For each iteration, block type 1 is computed first, then the 2s and 3s in parallel and finally all of the 4s in parallel as indicated by the coloured blocks in Figure 20.

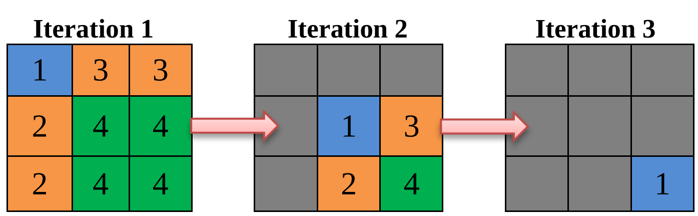


Figure 20: OpenCL GPU Execution of the LUD Algorithm (Mult_SMs_v2)

Mult_SMs_v2 (each OpenCL workgroup computing a different block) was more promising performance wise. It also made more sense to have the block to be computed residing in local memory of a single SM than to be segmented across multiple SMs. But using workgroups in this manner, we are finally able to beat the C++ Blocked LUD algorithm's runtime.

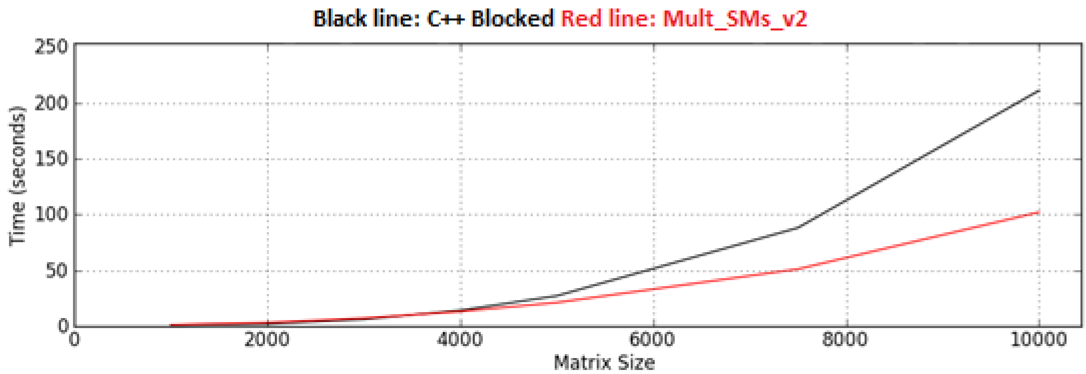


Figure 21: C++ Blocked vs Mult_SMs_v2 – optimal Block Size of 250 was used

Table 2: Data points of Figure 21

Matrix Size	C++ Blocked	Mult_SMs_v2
1000	1	1
2000	2	3
3000	6	7
4000	14	13
5000	27	21
7500	88	51
10000	211	102

Figure 22 shows the Block Size test for Mult_SMs_v2. The results indicate we are able to fully utilize the limit of 256 work items imposed on a single SM on our GPU.

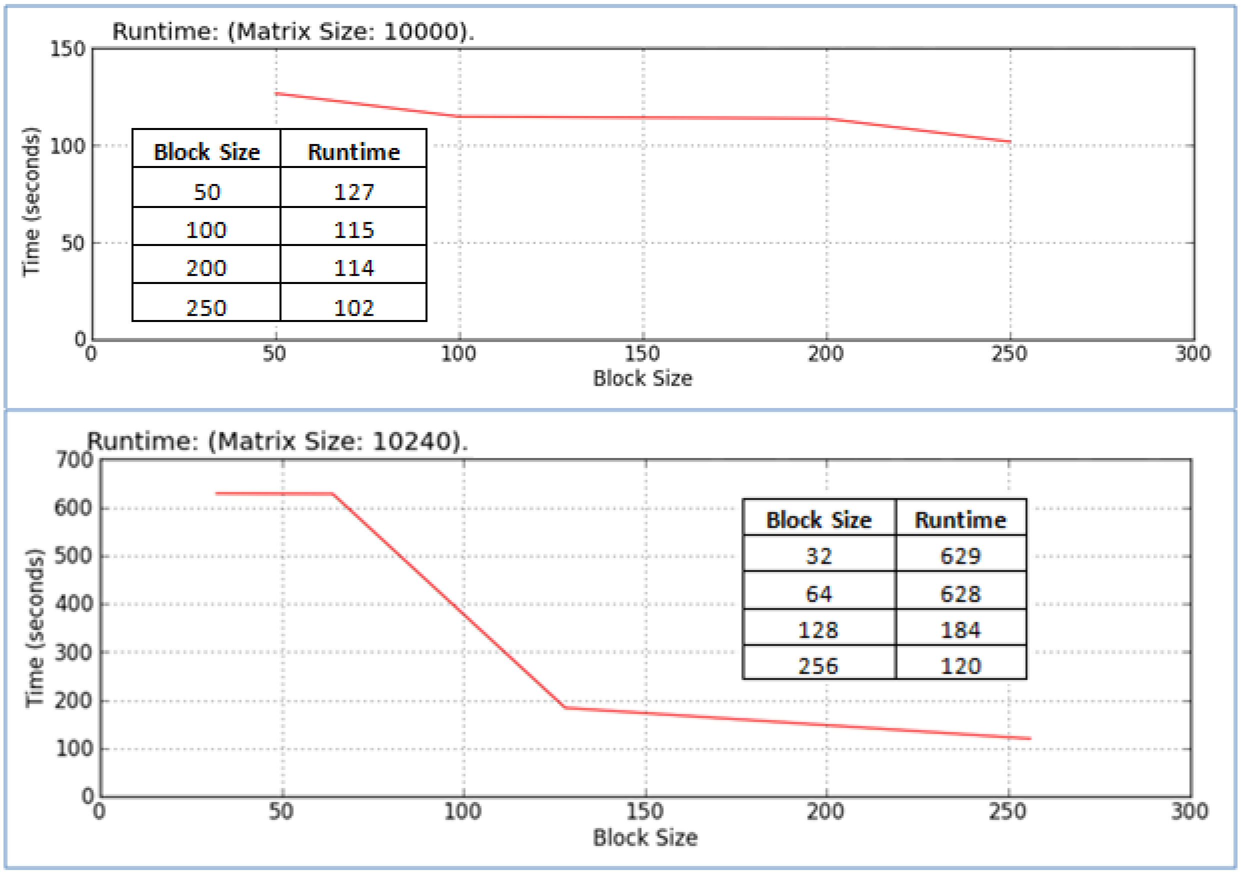


Figure 22: Best block size for Mult_SMs_v2 is 250 for a matrix of 10,000 and 256 for a matrix of 10,240

5.6. Adding Local Memory to Mult_SMs_v2

We used local memory for Mult_SMs_v2 and gained a slight performance improvement (Figure 24). Unfortunately, we were restricted to a block size of 50x50 due to the limited 32kb of local memory. This reduced the amount of parallelism that we were able to exploit within the SM (50 concurrent threads versus the previous 250). The block to be computed and the ***required*** (unlike before) portion of the dependent top block and/or left block was copied into local memory, as shown in Figure 24.

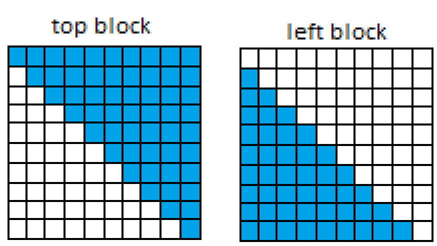


Figure 23: Required portion of a dependent block (in blue) was copied to local memory

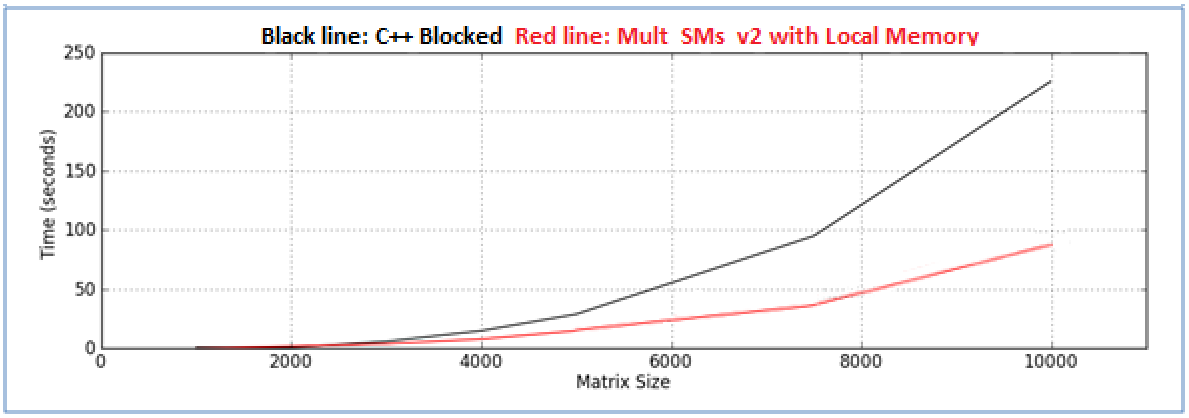


Figure 24: C++ Blocked vs Mult_SMs_v2 with Local Memory –Block Size of 50 was used

Table 3: Data points of Figure 24

Matrix Size	C++ Blocked	Mult_SMs_v2 with Local Memory
1000	1	0
2000	1	2
3000	6	4
4000	15	8
5000	29	15
7500	95	38
10000	226	89

This time for Mult_SMs_v2, using local memory showed a consistent improvement in performance.

6. Project Summary

We end the report by summarizing our results and how well we have met the project requirements and objectives. We also discuss lessons learned and future work that can be done.

6.1. Meeting Our Functional Requirements (author R. Rashid, E. Nasiri)

The table below provides a summary of all of our project deliverables. The numbers 1 to 6 correspond to the deliverables defined in Section 1.2.1: Functional Requirements. The requirements as they appear in Table 4 were defined in our Project Proposal's Validation and Acceptance Tests which is included in Appendix D.

Table 4: Summary of our Project Deliverables – ✔= Successful; × = Not Started

Requirements	1	2	3	4	5	6
Functionally Correct?	✔	✔	✔	✔	✔	×
Beats C++ Blocked runtime?	✔	✔	✔	✔	✔	×

Our C++ Blocked code beats our Non-Blocked code (Figure 7). The float16 OpenCL LUD code optimized for the CPU beats the runtime of the C++ Blocked code (Figure 14). Finally, the latest version of our GPU optimized LUD algorithm has the best runtime of all algorithms (89 seconds for a 10,000 x 10,000 matrix). Table 5 summarizes the performance of our algorithms and compares them to Intel's MKL LUD routine and dedicated hardware designed by Wei Zhang [10]. The Gigaflop calculation was done using the following formula:

$$\text{GFLOPS} = \frac{\frac{2}{3} \times n^3}{run\!-\!time} \text{ ; where } n = \text{matrix size}$$

Table 5: Performance Comparisons of Various LUD Algorithms

Platform	Clock Frequency	Algorithm	GFLOPS
CPU: Intel Core i7-2600K	3.4 GHz	C++ Blocked Code	3.02
CPU: Intel Core i7-2600K	3.4 GHz	OpenCL CPU-Optimized	1.0
CPU: Intel Core i7-2600K	3.4 GHz	OpenCL float16-Optimized	5.01
GPU: AMD Radeon 6900	880 MHz	OpenCL Multiple SM v.2	6.54
GPU: AMD Radeon 6900	880 MHz	OpenCL Multiple SM v.2 with Local Memory	7.5
CPU: MKL on Xeon 5160	3 GHz	MKL LUD Routine	42
FPGA: Stratix III 3SL340F1760C3	200 MHz	Dedicated Verilog Design by Wei Zhang [10]	47

6.2. Meeting Our Objectives (author: R. Rashid)

Asides from optimizing the runtime of our OpenCL algorithms, two of our objectives were to:

1. Minimize the amount of recoding required to move from one platform to another.
2. Record how much recoding is required for multi-platform compatibility.

Our OpenCL setup code (the CPU/GPU switching code discussed in Section 5.2) works with both AMD and Intel OpenCL 1.1 SDKs without the need for any code changes. Switching from CPU to GPU is as simple as changing the value of a single macro in a header file. We also provide the option of allowing the user select the compute device during runtime.

The OpenCL framework makes it relatively easy to write cross-platform code. However, code that is optimized for one platform is most likely not going to perform at optimum on another platform. Refer to Section 6.3 for a discussion of this.

6.3. Lessons Learned (author: E. Nasiri)

1) OpenCL Language

OpenCL is a very good platform for writing code where compute units need to work in parallel. There is a steep learning curve initially but the available documentation is thorough. It is important to understand that it is easy to develop an OpenCL code that has the same performance or worse performance than a similar C code that was optimized by the compiler to run on a CPU. This means that a beginner in OpenCL should not presume she can develop a much faster code just because she is using OpenCL.

2) User's Knowledge of Hardware

OpenCL tries to hide some of the complexities of hardware and parallel execution from the user and does so very well. However that does not imply that the developer doesn't need to know the target hardware for which she is coding. Knowing the architecture of processing units, memory hierarchy and data transfer bandwidth are crucial things to know in order to develop an optimal code.

3) Architecture Dependent Code

As mentioned above, the user needs to be aware of the hardware and since devices such as the CPU and GPU have fundamental architectural differences, if the user intends to produce a fast code, she must have the target platform in mind when developing the code. OpenCL promises platform-independence, meaning the same OpenCL code (with some minor tweaking) can be compiled to run on different devices (such as CPU, GPU, and FPGA). However, even though the code that was developed with CPU in mind as the target device can be compiled and run on the GPU, it will have a performance far from optimal on the GPU (and vice versa). The graph below shows the run-time of one of our CPU optimized OpenCL algorithms run on CPU and GPU. Compare the red curve (GPU run) and the green curve (our best OpenCL implementation for GPU). The difference in performance is very large.

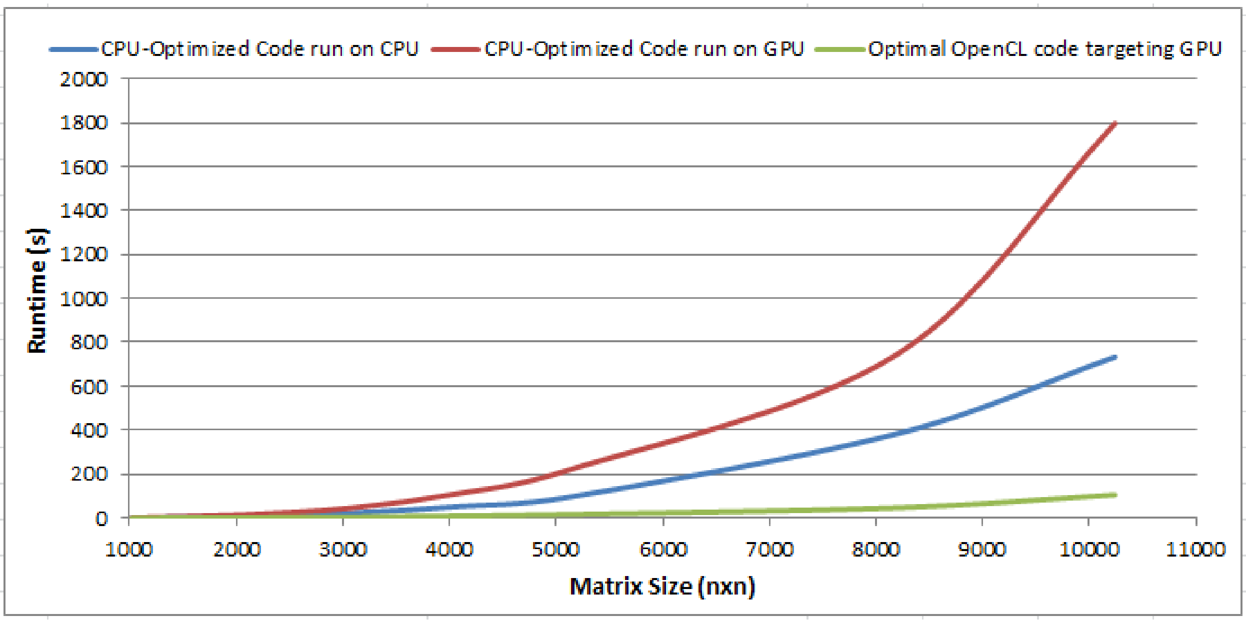


Figure 25: Optimal code for 1 platform is likely not going to perform optimally on another

6.4. Future Work (author: E. Nasiri, R. Rashid)

FPGA: The major item that can be considered for future work is taking measurements of our GPU-optimized OpenCL code on a FPGA. As part of our project we had access to a DE4 board with Altera's StratixIV FPGA on it. However, as confirmed by Altera's staff, the board had a voltage drop issue when large amount of resources were instantiated on the FPGA. Since our algorithm would try to exploit most of the resources of the FPGA, we were not able to perform runtime measurements on the FPGA.

OpenCL Memory Allocation Constraint: OpenCL restricts the amount of memory one can allocate for a target device. Due to this restriction, we were only able to run our algorithm for matrices up to the allowed size (close to 11,000×11,000). As a future work, it is possible to extend the algorithm to work for matrices larger than that by breaking the matrix down to sub-matrices (of maximum 11,000×11,000 size) and perform LU-Decomposition on each and perform extra computations required to obtain the final result (similar to blocking).

Using float16 Data Type on GPU Device: We store the matrix in a 1D array of float data type. On the CPU, we are able to treat a row of a block as float16 data type and perform operations on it. Due to stricter OpenCL syntax requirements on the GPU, we were unable to write the same code for the GPU. To use float16 and take advantage of **SIMD SSE** instructions, it was required to store the matrix in a float16 array instead. This effort was initiated but never realized due to time constraints. The entire algorithm (host file as well as the OpenCL kernels) would have required a major rewrite.

More Parallelism on a GPU when Using Local Memory: As explained in the Section 5.6 , we are restricted to a block size of 50 due to the 32kb of local memory. This limits the number of work-items to 50 (limit is 256) and workgroups to 1 on a SM because we coded the algorithm to have 1 thread operate on a single column. Recoding the algorithm such that we have 256 threads, with block size 16 will also allow multiple workgroups to exist within a single SM (around 10 ≈ 32kb / (≈3 blocks)*(16*16)*(4 bytes/float). Workgroups are computed independently so if one workgroup is stalled to meet a synchronization point, the SM can quickly switch to another workgroup.

References

[1] William H. Press. 2007. *Numerical recipes: the art of scientific computing.* (3rd edition). p48.

[2] L.W. Nagel, D. O. Pederson. 12 April 1973. "*SPICE (Simulation Program with Integrated Circuit Emphasis)*". [Online]. Accessed August 15, 2011. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1973/ERL-382.pdf

[3] Thomas Linwood Quarles. "Analysis of Performance and Convergence Issues for Circuit Simulation". Section 2.1.2. p16. [Online]. Accessed August 15, 2011. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1989/ERL-89-42.pdf

[4] Thomas Telkamp. "Best Practices for Determining the Traffic Matrix in IP Networks". *Apricot (Asia Pacific Regional Internet Conference on Operational Technologies).* [Online]. Accessed September 2, 2011. http://www.apricot.net/apricot2005/slides/C5-4_3.pdf

[5] "The Stoichiometry of Reactions – Introduction". University of Wisconsin. [Online] Accessed September 2, 2011. http://jbrwww.che.wisc.edu/home/jbraw/chemreacfun/ch2/slides-stoi-2up.pdf

[6] "The Leontief Open Production Model or Input-Output Analysis". College of Redwoods. [Online] Accessed September 8, 2011. http://online.redwoods.cc.ca.us/instruct/darnold/laproj/fall2001/iris/lapaper.pdf

[7] GNU Scientific Library. LU-Decomposition Reference Manual. [Online]. Accessed September 10, 2011. Available: http://www.gnu.org/s/gsl/manual/html_node/LU-Decomposition.html

[8] Intel. Intel Math Kernel Library. [Online] Accessed September 11, 2011. Available: http://software.intel.com/en-us/articles/intel-math-kernel-library-purchase/

[9] AMD. AMD OpenCL SDK. [Online]. Accessed September 11, 2011. Available: http://developer.amd.com/sdks/amdappsdk/downloads/pages/default.aspx

[10] Wei Zhang, Vaughn Betz, Jonathan Rose. *"Portable and Scalable FPGA-Based Acceleration of a Direct Linear System Solver"*. 8-10 Dec. 2008.

[11] Mac Research. OpenCL Tutorial – Memory Access and Layout. [Online]. Accessed November 3, 2011. Available: http://www.macresearch.org/opencl_episode4

[12] Sam Siewert, "Algorithm Acceleration Using Single Instruction Multiple Data", Intel. [Online]. Accessed November 10, 2011. Available: http://software.intel.com/en-us/articles/using-intel-streaming-simd-extensions-and-intel-integrated-performance-primitives-to-accelerate-algorithms/

Appendix A: Glossary **(author: E. Nasiri)**

Blocked Algorithm

Blocked algorithm involves breaking down a large matrix into smaller chunks (called "blocks") and performing computations on each block separately. In the context of matrix manipulation, since the blocks are not independent, in order for the blocked algorithm to produce the correct output, some extra work must be done (overhead). If the algorithm can be implemented in a way where the speed-up from executing blocks in parallel is higher than the overhead, then the blocked algorithm will be faster than the non-blocked equivalent.

Dense Matrix

A dense matrix is a matrix which has very few 0 entries. In contrast, sparse matrices contain mostly 0 entries and very few non-zero entries. There are certain tricks that can be employed to speed up the LU-Decomposition algorithm for sparse matrices. For our project, we will not worry about this scenario.

float16

float16 is a built-in data type in defined in OpenCL. It allows the programmer to define 16 floating point numbers as a group and operate on them utilizing the Single Instruction Multiple Data (for example, add two sets of 16 floats at the same time in parallel).

FPGA

Field Programmable Gate Arrays (FPGAs) are integrated logic circuits that can be dynamically programmed to implement any kind of hardware logic the user desires. Hardware Description Languages (HDLs) such as Verilog are used to program FPGAs. Altera's OpenCL complier takes a piece of OpenCL code and implements it in hardware on the FPGA.

LU-Decomposition

LU-Decomposition is the algebraic operation of factorizing a matrix into a lower triangular (L) and an upper triangular matrix (U). LUD is one of the fundamental methods for solving a system of n linear equations and n unknowns.

Matrix A = Lower Triangular x Upper Triangular

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & -4 & 6 \\ 3 & -9 & -3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & -8 & 0 \\ 3 & -15 & -12 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Non-Blocked Algorithm

Non-blocked algorithm refers to an algorithm that takes an entire matrix and performs an operation on the matrix. These algorithms usually run into run-time inefficiency once the size of the matrix become considerably large. See Blocked algorithm for contrast.

OpenCL (Open Computing Language)

OpenCL is a language that has been recently developed by Khronos group which consists of a group of companies that includes Apple, Intel, AMD, Altera and Texas Instruments. This language is the first open and free standard language for cross-platform parallel programming.

SIMD (Single Instruction operating on Multiple Data simultaneously)

On Intel architectures, Streaming SIMD Extensions (SSE) instructions are used to realize SIMD operations on single precision floating point data (floats).

Single Precision Floating Point Format

Single precision floating point format is a standard IEEE format in which a floating point number is stored in 4 bytes (32 bits) of memory. For our purposes, it is less precise, takes up less storage and has better performance than the Double Precision Floating Point format.

Streaming Multiprocessors (SMs)

SMs are discrete execution units on a GPU with their own private memory, pipeline and functional units including ones that can perform SIMD floating point operations. Several streaming processors with their own register files the execute instructions.

Workgroups and Work-Items

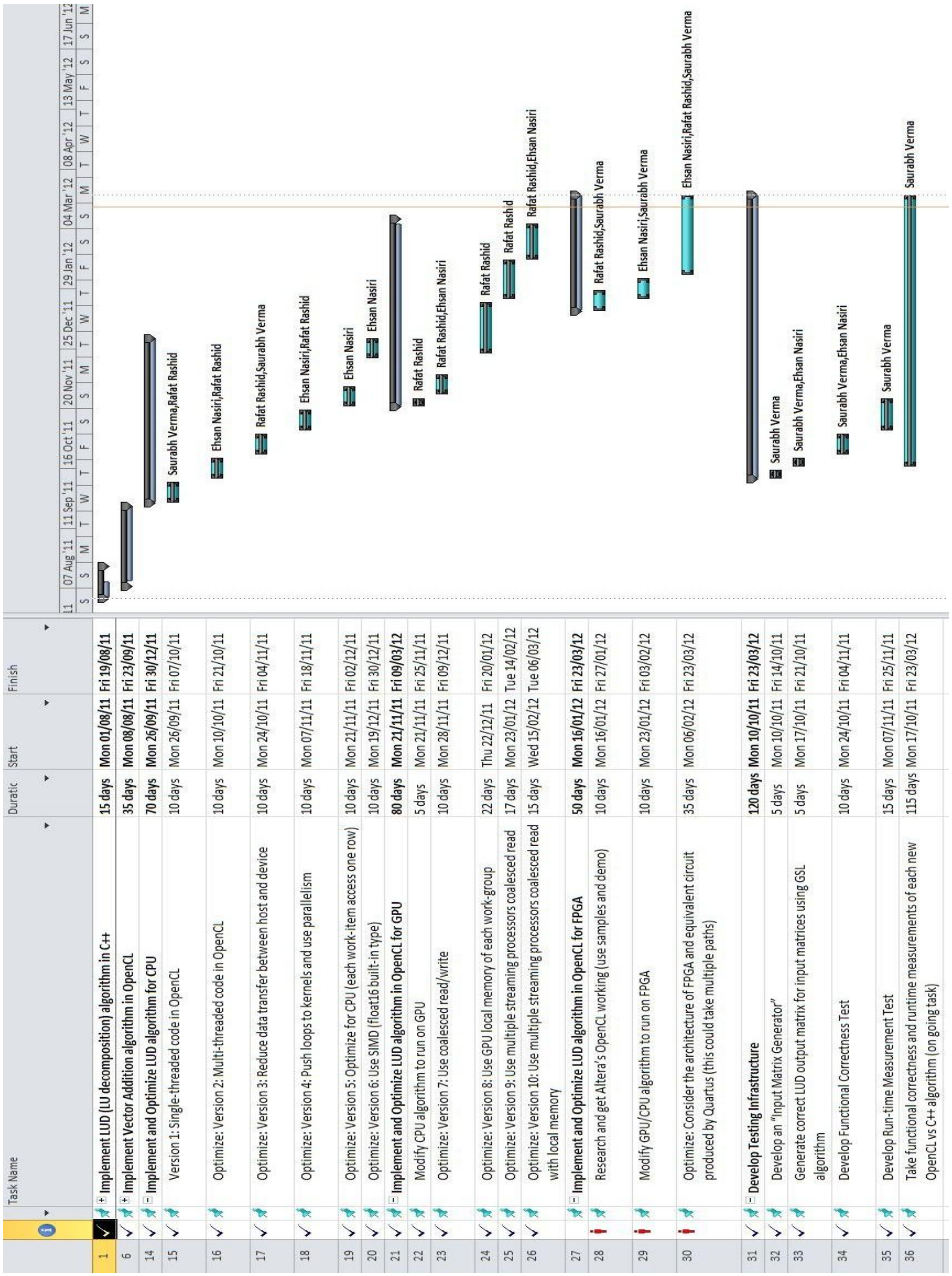
In OpenCL, work-items are threads (with private memory) and workgroups are a group of threads (with a larger shared local memory and get allocated to the same execution unit).

Appendix B: Final Work Breakdown Structure (author S. Verma)

#	Task	Ehsan	Rafat	Saurabh
1	**Implement LUD algorithm in C++**			
2	Literature research on LUD algorithms	A		R
3	Implement non-blocked LUD algorithm in C++	R		
4	Implement LUD algorithm using GNU Scientific Library in C++ (GSL)	R		
5	Implement blocked LUD algorithm in C++	A	R	
6	**Implement Vector Addition algorithm in OpenCL**			
7	Study basic OpenCL concepts	A	R	A
8	Setup OpenCL SDK			
9	• Research on OpenCL SDK's for CPU and GPU	R		
10	• Setup OpenCL SDK's on CPU (Intel) and GPU (AMD)			R
11	Write Vector Addition algorithm using single thread on AMD SDK		R	
12	Write OpenCL algorithm that works on both AMD and Intel SDK			R
13	Modify Vector Addition to use multiple threads	R	A	
14	**Implement and Optimize LUD algorithm in OpenCL for CPU**			
15	Version 1: Single-threaded code in OpenCL		A	R
16	Optimize: Version 2: Multi-threaded code in OpenCL	R	A	
17	Optimize: Version 3: Reduce data transfer between host and device		R	A
18	Optimize: Version 4: Push loops to kernels and use parallelism	R	A	
19	Optimize: Version 5: Optimize for CPU (each work-item access one row)	R		
20	Optimize: Version 6: Use *SIMD* (float16 built-in type)	R		
21	**Implement and Optimize LUD algorithm in OpenCL for GPU**			
22	Modify CPU algorithm to run on GPU		R	
23	Optimize: Version 7: Use coalesced read/write	A	R	
24	Optimize: Version 8: Use GPU local memory of each work-group		R	
25	Optimize: Version 9: Use multiple streaming processors		R	
26	Optimize: Version 10: Try using float16 on GPUs	R	A	
27	**Implement and Optimize LUD algorithm in OpenCL for FPGA**			
28	Research and get Altera's OpenCL working (use samples and demo)		A	R
29	Modify GPU/CPU algorithm to run on FPGA	R		A
30	Optimize: Consider the architecture of FPGA and equivalent circuit produced by Quartus (this could take multiple paths)	R	A	A
31	**Develop Testing Infrastructure**			
32	Develop an "Input Matrix Generator"			R
33	Generate correct LUD output matrix for input matrices using GSL algorithm	A		R
34	Develop Functional Correctness Test	A		R
35	Develop Run-time Measurement Test			R
36	Take functional correctness and runtime measurements of each new OpenCL vs C++ algorithm (on going task)			R

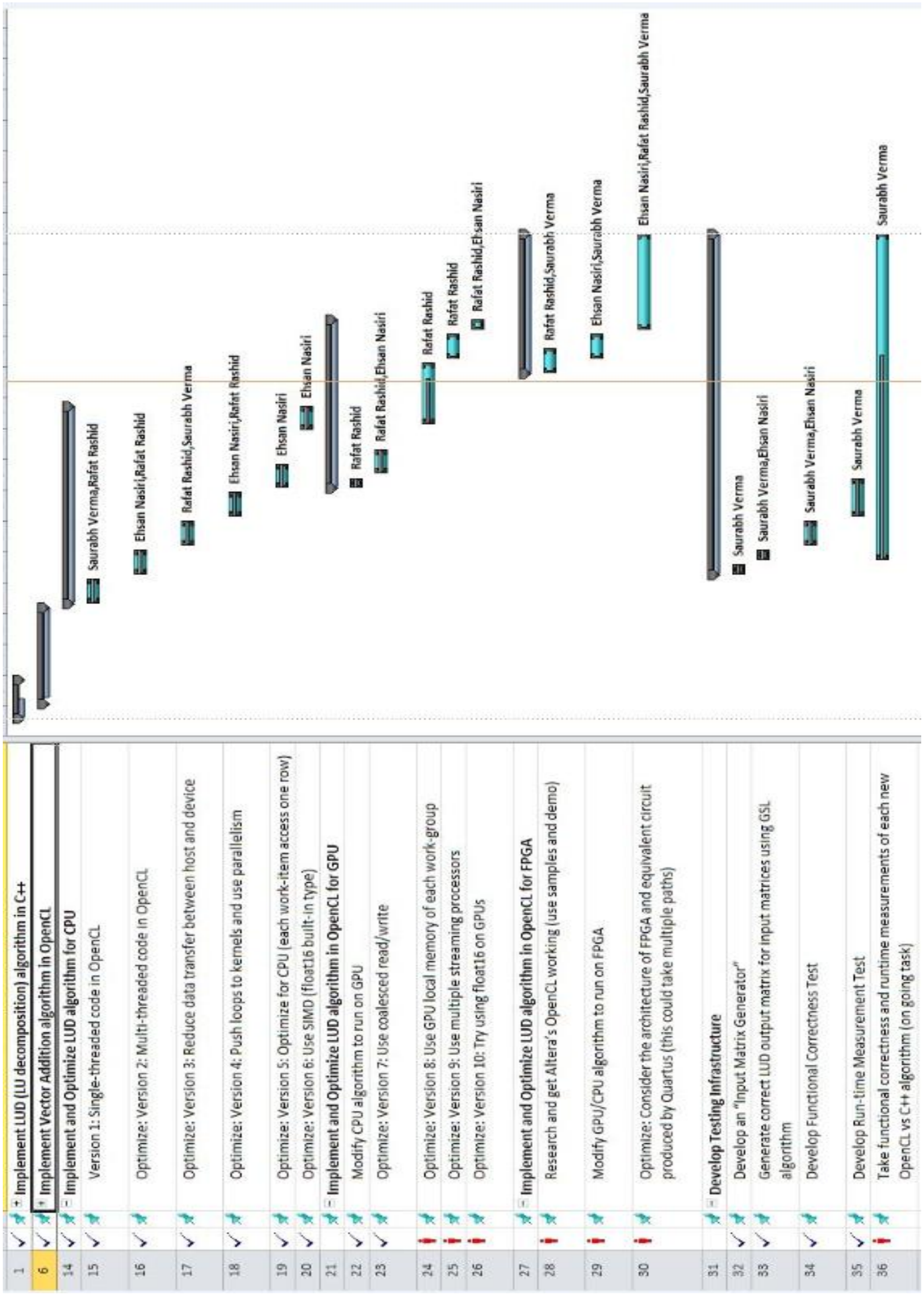
Appendix C: Gantt Chart History **(author: S. Verma)**

Our final gantt chart as of March 22, 2012 – everything complete except the FPGA milestone:

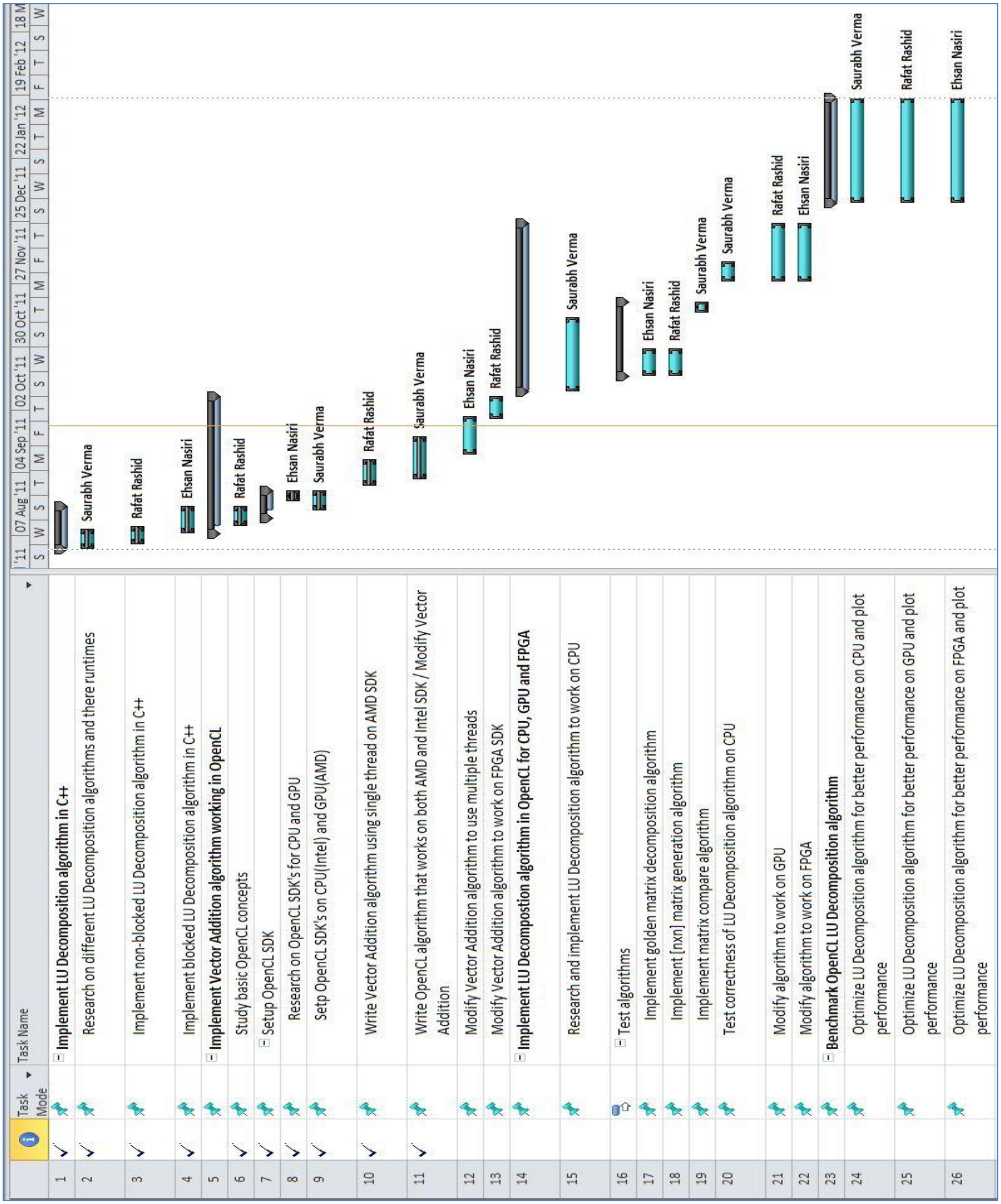


	Task Name	Duration	Start	Finish
1	Implement LUD (LU decomposition) algorithm in C++	15 days	Mon 01/08/11	Fri 19/08/11
6	Implement Vector Addition algorithm in OpenCL	35 days	Mon 08/08/11	Fri 23/09/11
14	Implement and Optimize LUD algorithm for CPU	70 days	Mon 26/09/11	Fri 30/12/11
15	Version 1: Single-threaded code in OpenCL	10 days	Mon 26/09/11	Fri 07/10/11
16	Optimize: Version 2: Multi-threaded code in OpenCL	10 days	Mon 10/10/11	Fri 21/10/11
17	Optimize: Version 3: Reduce data transfer between host and device	10 days	Mon 24/10/11	Fri 04/11/11
18	Optimize: Version 4: Push loops to kernels and use parallelism	10 days	Mon 07/11/11	Fri 18/11/11
19	Optimize: Version 5: Optimize for CPU (each work-item access one row)	10 days	Mon 21/11/11	Fri 02/12/11
20	Optimize: Version 6: Use SIMD (float16 built-in type)	10 days	Mon 19/12/11	Fri 30/12/11
21	Implement and Optimize LUD algorithm in OpenCL for GPU	80 days	Mon 21/11/11	Fri 09/03/12
22	Modify CPU algorithm to run on GPU	5 days	Mon 21/11/11	Fri 25/11/11
23	Optimize: Version 7: Use coalesced read/write	10 days	Mon 28/11/11	Fri 09/12/11
24	Optimize: Version 8: Use GPU local memory of each work-group	22 days	Thu 22/12/11	Fri 20/01/12
25	Optimize: Version 9: Use multiple streaming processors coalesced read	17 days	Mon 23/01/12	Tue 14/02/12
26	Optimize: Version 10: Use multiple streaming processors coalesced read with local memory	15 days	Wed 15/02/12	Tue 06/03/12
27	Implement and Optimize LUD algorithm in OpenCL for FPGA	50 days	Mon 16/01/12	Fri 23/03/12
28	Research and get Altera's OpenCL working (use samples and demo)	10 days	Mon 16/01/12	Fri 27/01/12
29	Modify GPU/CPU algorithm to run on FPGA	10 days	Mon 23/01/12	Fri 03/02/12
30	Optimize: Consider the architecture of FPGA and equivalent circuit produced by Quartus (this could take multiple paths)	35 days	Mon 06/02/12	Fri 23/03/12
31	Develop Testing Infrastructure	120 days	Mon 10/10/11	Fri 23/03/12
32	Develop an "Input Matrix Generator"	5 days	Mon 10/10/11	Fri 14/10/11
33	Generate correct LUD output matrix for input matrices using GSL algorithm	5 days	Mon 17/10/11	Fri 21/10/11
34	Develop Functional Correctness Test	10 days	Mon 24/10/11	Fri 04/11/11
35	Develop Run-time Measurement Test	15 days	Mon 07/11/11	Fri 25/11/11
36	Take functional correctness and runtime measurements of each new OpenCL vs C++ algorithm (on going task)	115 days	Mon 17/10/11	Fri 23/03/12

Gantt chart submitted with the Group Progress document (as of January 17, 2012):



Gantt chart submitted with the Final Project Proposal (as of October 18, 2011):



Appendix D: Validation and Acceptance Tests

As outlined in our Project Proposal, we have developed automated tests to verify and evaluate our final design against the project requirements and metrics. The test specification provided below remains unchanged since the submission of our Final Project Proposal.

Functional Correctness Test

Test Objective	To validate the mathematical results of the LU-Decomposition algorithm on each of the target platforms.
Verification Procedure	Compare the generated L and U matrices with the corresponding correct L and U set.
Acceptable Outcome	The solution is considered valid if the L and U matrices produced by the LU-Decomposition algorithm are correct. A variance of +/-5% in the normalized Euclidian distance between produced LU and correct LU matrices will be accepted to account for the target platform's inherent limitation in performing floating-point arithmetic operations.
Resources Needed	A large set of correct L and U matrices of varying sizes will need to be generated with the aid of a known working version of a program that performs LU-Decomposition. We will use the GNU Scientific Library (GSL) to produce the correct L and U matrices.

Run-time Efficiency Test

Test Objective	To measure the run-time efficiency of the OpenCL LU-Decomposition algorithm.
Verification Procedure	Compare the run time of the OpenCL LU-Decomposition algorithm with that of the serial C/C++ version of the algorithm which will execute on the system's CPU.
Acceptable Outcome	The algorithm will be accepted if it runs faster than the serial C/C++ version of the algorithm.
Resources Needed	Working version of the serial C/C++ LU-Decomposition algorithm.

Appendix E: Non-Blocked LUD Algorithm (author S. Verma)

This algorithm performs two operations on the matrix elements:

1. Division of all the elements below the diagonal in the column, $a_{k+1,k}$ to $a_{N,k}$, by the diagonal element, $a_{k,k}$.
2. Multiplication of column elements, $a_{k+1,k}$ to $a_{N,k}$ by row element, $a_{k,j}$, and the subsequent subtraction of the result from the column elements below the row element, $a_{k+1,j}$ to $a_{N,j}$. The multiplication and subtraction is repeated for j from k+1 to N. All the operations are repeated for the next diagonal element until the last diagonal element is reached.

Pseudo-code Non-Blocked LU Decomposition Algorithm

for $k = 1$ to $N - 1$ **do** /*for each diagonal element*/
  for $i = k + 1$ to N **do** /*for each element below it*/
    $a_{i,k} \leftarrow a_{i,k} / a_{k,k}$ /*normalize*/
  end for
  for $j = k + 1$ to $N - 1$ **do** /*for each column right of current diagonal element*/
    for $i = k + 1$ to N **do** /*for each element below it*/
      $a_{i,j} \leftarrow a_{i,j} - a_{i,k} \times a_{k,j}$
    end for
  end for
end for

Source: Wei Zhang's PhD thesis [10]

Appendix F: Blocked LUD Algorithm **(author: S. Verma)**

This algorithm divides the matrix into four block types. With this blocking method, there are four kinds of computations performed on the blocks:

Case 1: All three blocks (current, left-most, and top-most) are the same physical block

Case 2: The current block is the same as the left-most block

Case 3: the current block is the same as the top-most block

Case 4: all three blocks are different

Pseudo-code Blocked LU Decomposition Algorithm

Case 1:
same as simple LU factorization with N_b instead of **N** /*See Algorithm 1*/

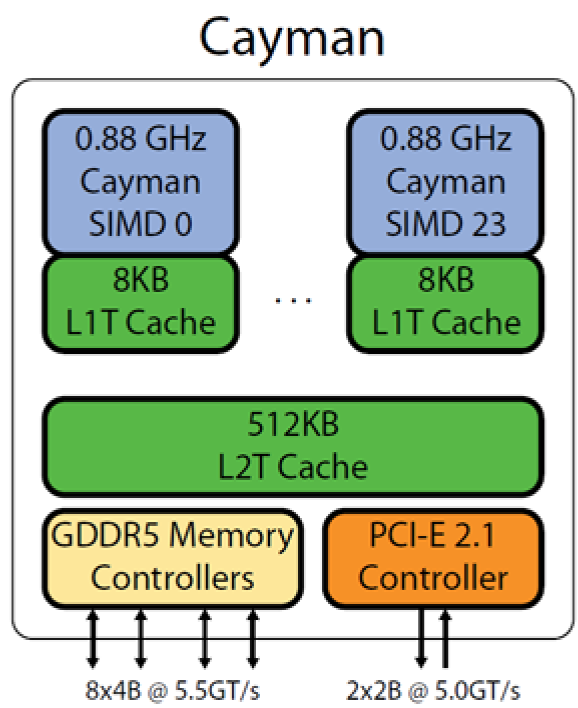
Case 2:
for $k = 1$ to N_b **do** /*for each diagonal element in top-most block (u)*/
    for $i = 1$ to N_b **do** /*for each element below it in current block (a)*/
        $a_{i,k} \leftarrow a_{i,k}/u_{k,k}$ /*normalize*/
    end for
    for $j = k + 1$ to N_b **do** /*for each column right of current diagonal element*/
        for $i = 1$ to N_b **do** /*for each element below it in current block (a)*/
            $a_{i,j} \leftarrow a_{i,j} - a_{i,k} \times u_{k,j}$
        end for
    end for
end for

Case 3:
for $k = 1$ to N_b **do** /*for each column in left-most block (l)*/
    for $j = 1$ to N_b **do** /*for each column in current block (a)*/
        for $i = k + 1$ to N_b **do** /*for each element below it*/
            $a_{i,j} \leftarrow a_{i,j} - l_{i,k} \times a_{k,j}$
        end for
    end for
end for

Case 4:
for $k = 1$ to N_b **do** /*for each column in left-most block (l)*/
    for $j = 1$ to N_b **do** /*for each column in top-most block (u)*/
        for $i = 1$ to N_b **do** /*for each element below it in current block (a)*/
            $a_{i,j} \leftarrow a_{i,j} - l_{i,k} \times u_{k,j}$
        end for
    end for
end for

Appendix G: Glados' CPU and GPU Specifications (author: S. Verma)

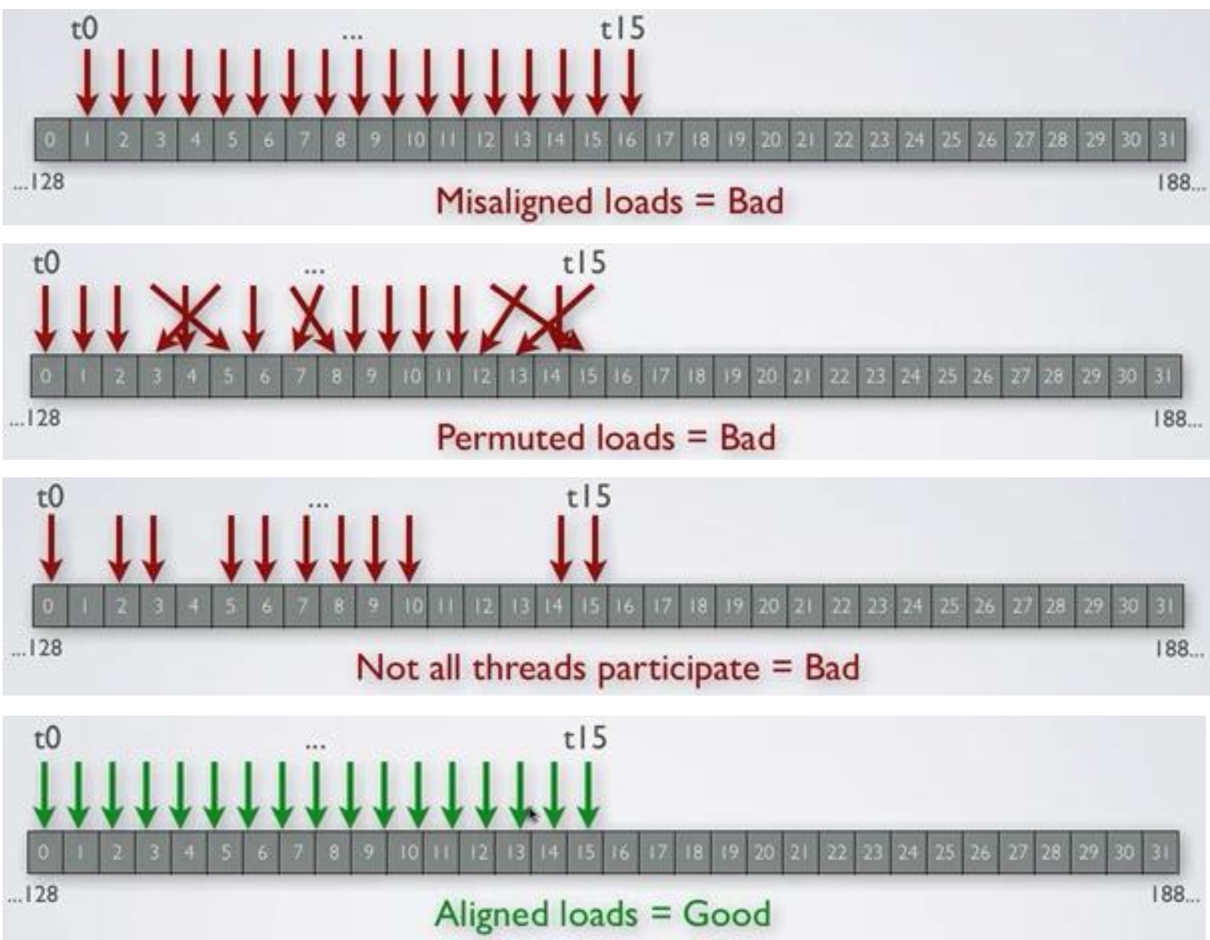
CPU	GPU
Intel (R) Core(TM) i7-2600K @ 3.4GHz Global Memory (MB): 2047 Max Work Group Size: 1024 Number of Cores: 4 Number of Parallel Compute Cores (threads): 8 L1 Cache size: 4x32 KB instruction caches 4x32 KB data caches L2 Cache size: 4x256 KB L3 Cache size: 8 MB shared cache	AMD Radeon HD 6900 - Cayman Global Memory (MB): 2048 Global Memory Cache (MB): 0 Max Clock (Mhz): 880 Local Memory (KB): 32 Max Work Group Size: 256 Number of parallel compute cores: 24 L1 Cache size: 8 KB L2 Cache size: 512 KB shared cache



Appendix H: Aligned Coalesced Accesses **(author: R. Rashid)**

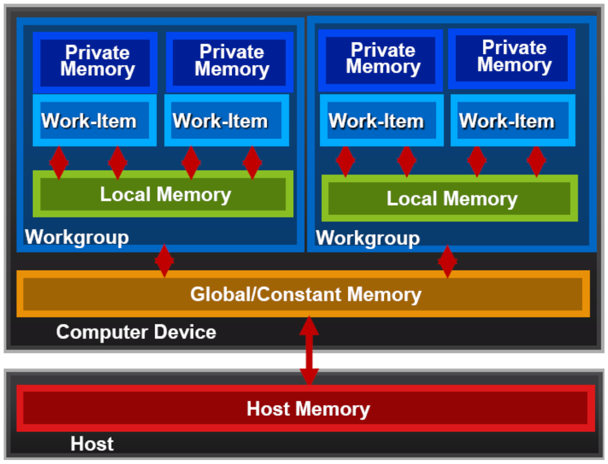
Coalesced read/write corresponds to each kernel (thread) accessing shared data column wise instead of sequentially (or row wise). In the diagrams below, each arrow corresponds to a memory access by a different thread. Each element is a float and 4 bytes wide. Assume the matrix block size is 16x16. Assume the half-warp (group of consecutive threads that are executed together on the device) is 16. Then a memory access at address 128 is aligned (128 / (4*16) = whole number) but 132 is not.

Diagram 4 is an example of an aligned coalesced read. The GPU hardware recognizes this and will grab the 64 byte block starting at address 128 at once instead of in 16 separate instances which most likely will be the case with Diagrams 1-3. This is the reason a coalesced read is much faster than each thread sequentially accessing multiples of 16 bytes on a GPU. Refer to *http://www.macresearch.org/opencl_episode4* [11] which provides a 1 hour video tutorial on OpenCL and shared memory access and layout architecture of GPUs.

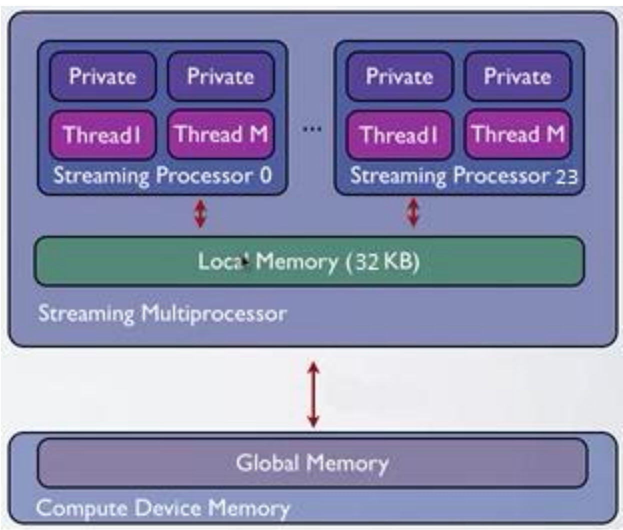


Appendix I: Memory Layout of OpenCL and GPUs **(author: R. Rashid)**

OpenCL assumes a memory hierarchy as shown on the right. In brief, there are two components, 1) the host and 2) the kernels (threads/work-items) that are managed by the host.



A typical GPU is composed of many streaming multiprocessors (24 on our target machine, Glados). Each SM in turn is composed of several streaming processors (SP) and for our purposes, 1 or more floating point functional units capable of performing computation. The 32kb of Local Memory (which is much faster and smaller than Global Memory) is shared between these streaming processors. The memory layout for a SM is illustrated in the diagram on the left.



Our GPU Memory Layout

Currently, for each matrix block, our algorithm 1) copies required data from the matrix buffer stored in the Global Memory to the Local Memory of a single SM 2) performs the computation in parallel on a single SM and 3) applies the changes to the Global Memory after the computation is done.

Appendix J: Taking Bank Conflicts into Consideration (author: R. Rashid)

GPU memory is organized into memory banks. If two threads access elements from the same bank, these accesses will be serialized. The diagram below provides a visual representation.

```
Bank    |      1      |      2      |      3      |...
Address |  0  1  2  3 |  4  5  6  7 |  8  9 10 11 |...
Address | 64 65 66 67 | 68 69 70 71 | 72 73 74 75 |...
...
```

In the diagram, the bytes 0-3, 64-67, … fall within the same bank. On our AMD GPU, we have 32 banks which are interleaved with a granularity of 32-bits. So if each thread in a half-warp (see Appendix H) accesses successive 32-bits, there will be no bank conflicts. Fortunately, a float (used to represent a single matrix element) is 4 bytes.

As an example, this snippet from the CPU-optimized LUD algorithm has many bank conflicts:

```
const int i = get_global_id(0); //thread-id

for (uint j=k+1; j<blkSize; j++)
     if (i>=k+1 && i<blkSize)
          block[i*blkSize+j] -= (block[+i*blkSize+k] * block[k*blkSize+j]);
```

As explained in Appendix H, we modified the memory access pattern to be coalesced as shown in the diagram below. This also removes bank conflicts:

```
const int j = get_global_id(0);

for (uint i=k+1; i<blkSize; i++)
     if (j>=k+1 && j<blkSize)
          block[i*blkSize+j] -= (block[i*blkSize+k] * block[k*blkSize+j]);
```