Multi-Platform LU-Decomposition Solution in OpenCL

Ehsan Nasiri, Rafat Rashid, Saurabh Verma

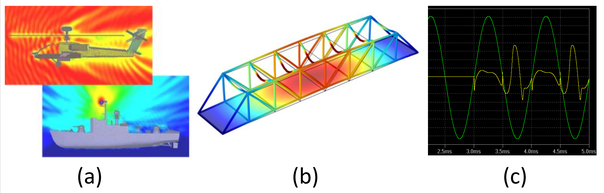
Supervisor: Vaughn Betz **Administrator:** Christiana Amza **Section:** 8 **Project ID:** 2011027

UNIVERSITY OF TORONTO

OpenCL

1. MOTIVATION

LU-Decomposition (LUD) is a basic algebraic operation with a wide range of applications.



(a) (b) (c)

Figure 1: (a) Electromagnetic solver simulation (b) Structure stress simulation (c) Spice Voltage simulation

LU-Decomposition is the most expensive part of solving a system of linear equations.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} \end{bmatrix} * \begin{bmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ 0 & U_{22} & U_{23} & U_{24} \\ 0 & 0 & U_{33} & U_{34} \\ 0 & 0 & 0 & U_{44} \end{bmatrix}$$

Figure 2. LU-Decomposition: A = L * U

Existing challenges:

- High computing complexity: $O(n^3)$
- Long computation/simulation times
- Existing solutions provided by Intel (MKL) and AMD (OpenCL SDK) are expensive and platform specific

2. OUR SOLUTION

Develop a high-performance code that can execute across heterogeneous devices without recompilation.

Test Hardware:

CPU:	Intel Core i7-2600K @ 3.4 GHz
GPU:	AMD Radeon HD 6900 Series (Cayman)
FPGA:	Stratix IV GX

OpenCL

- Free cross-platform parallel computing language
- Recently developed by the Khronos group
  - Apple, Intel, AMD, Altera, TI, etc.

3. GOALS AND OBJECTIVES

- **Functional Correctness:** Produce the correct L and U on each of the target platforms
- **Runtime Efficiency:** Beat runtime of serial C++ LUD implementation
- **Portability Across Devices:** Cross-platform LUD solution using OpenCL

4. SYSTEM DESIGN

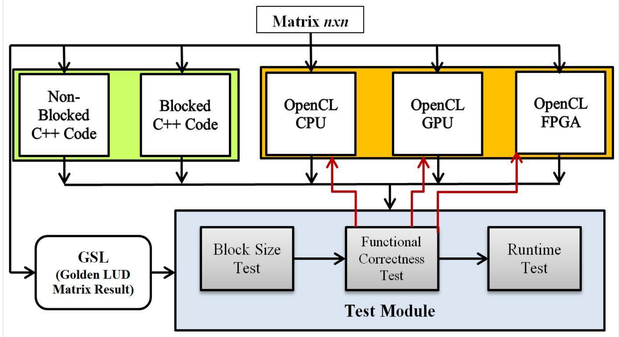


Figure 3: The System Block Diagram

4.1 BLOCKED ALGORITHM APPROACH

- Divide matrix into smaller blocks
- *Dependencies:* black block depends on top-most and left-most gray blocks

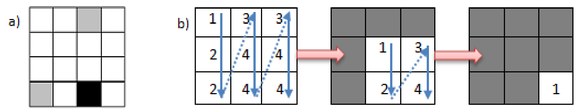


Figure 4: a) Block dependency b) Blocked computation

Why use Blocking?

- Operates on at most 3 blocks at any given time
- Correct block size takes advantage of caches, thus reducing the time required to fetch the data.

4.2 OPENCL LUD ALGORITHM DESIGN ON CPU

Our implementation on the CPU performs the computation as shown in Figure 4.b):

- Keep the CPU occupied by using parallelism within a block
- Optimal block size of 250 (with 250 concurrent threads) is used
- Utilizes CPU caches and SIMD instructions
- Dependencies between blocks impose a restriction on the level of parallelism we can achieve

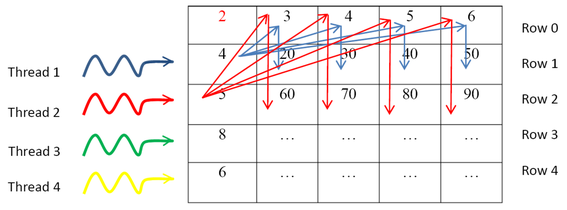


Figure 5: Parallelism in block processing

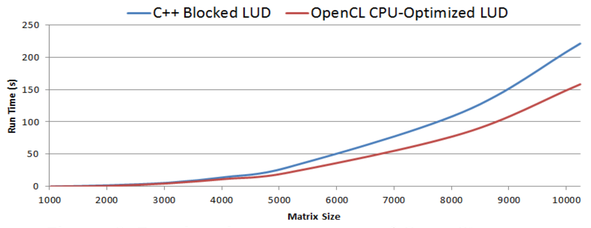


Figure 6. Runtime improvements of OpenCL

4.3 OPENCL LUD ALGORITHM DESIGN ON GPU

CPU to GPU Port → Single SM + Global Memory → Aligned Coalesced Data Access → Remove Bank Conflicts → Multiple SMs → Local Memory

Figure 7. OpenCL GPU Algorithm Design Process

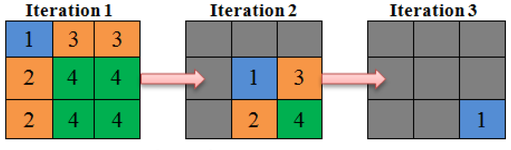


Figure 8: LUD Algorithm Execution on GPU

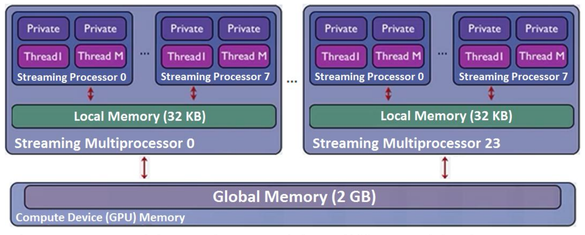


Figure 9: Provide each SM with a different block to compute

Two levels of parallelism exploited:

- Perform computations within a block in parallel (like it is done on the CPU)
- Compute multiple blocks in parallel (coloured blocks in Figure 8) by taking advantage of the Streaming Multiprocessors on a GPU

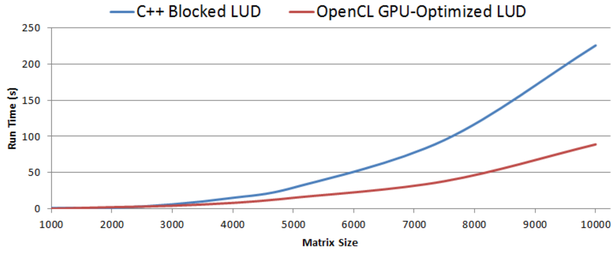


Figure 10. GPU algorithm beats target C++ Blocked algorithm

5. TESTING, VERIFICATION AND RESULTS

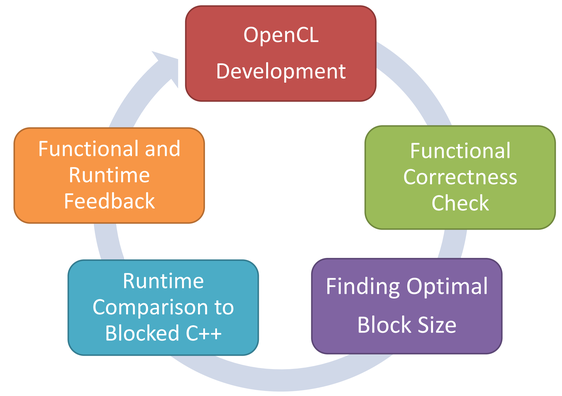
$$\text{GFLOPS} = \frac{2/3 \times (\text{matrix size})^3}{\text{runtime}}$$

Table 1. Gigaflop calculation for CPU, GPU and FPGA

LUD Algorithm	GFLOPs
C++ Non-Blocked	1
C++ Blocked	3.02
OpenCL CPU	5.01
OpenCL GPU	7.5

We successfully beat the runtime of the C++ Blocked code with our OpenCL CPU and GPU implementations.

Iterative Algorithm Design Process



6. CONCLUDING REMARKS

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.

7. FUTURE WORK

- Optimize algorithm for FPGA
- Utilize SIMD SSE instructions on GPU
- Extend algorithm to operate on matrices larger than 10k x 10k elements
- For a block size of m, launch m x m threads instead of the current m to exploit more parallelism

ACKNOWLEDGEMENTS

- Vaughn Betz (*Supervisor*)
- Christiana Amza (*Administrator*)
- Deshanand Singh (*Altera*)
- Alex Rodionov (*Altera*)