GPU DECLARATIVE FRAMEWORK: DEFG

Dissertation Defense

Robert Senser

October 29, 2014

PhD Committee:
Gita Alaghband (chair)
Tom Altman (advisor)
Michael Mannino
Boris Stilman
Tam Vu
Presentation Outline

• Motivation for *GPU Declarative Framework*: DEFG
• Background: Graphics Processing Units (GPUs) and OpenCL
• DEFG Framework
  – Description
  – Performance
• Diverse Applications using DEFG
  – Image Filters (Sobel and Median)
  – Breadth-First Search
  – Sorting Roughly Sorted Data
  – Iterative Matrix Inversion
• Dissertation Accomplishments
• Future Research
DEFG Motivation

• GPUs can provide high throughput
  – Radeon HD 7990: 2 TFLOPS (double-precision)
• Developing parallel HPC software is difficult
• Parallel development for GPUs is even more difficult
• GPU HPC software development requires:
  – Understanding of unique GPU hardware characteristics
  – Use of specialized algorithms
  – Use of GPU-specific, low-level APIs
• Driving notion behind DEFG: *Let software minimize the complexity and difficulty*
Background: GPUs and OpenCL

• Graphics Processing Unit (GPU)
  – Highly specialized coprocessor
  – Hundreds of cores, with thousands of threads
  – SIMT: *Single Instruction, Multiple Thread*
    • Similar to Single Instruction, Multiple Data (SIMD) model
    • Threads not on the execution path are paused

• Common GPU programming environments
  – OpenCL: an open, royalty-free standard
  – CUDA: NVIDIA proprietary

• DEFG is designed for OpenCL
High-Level GPU Architecture

GPU Characteristics:
• Processors commonly connected by Peripheral Component Interconnect Express (PCIe) bus
• GPU has own fast Global RAM
• Threads have a small amount of fast local memory
• May have a hardware cache
• Many hardware-managed threads
• Lacks CPU-style predictive branching, etc.
OpenCL Overview

• Specification maintained by Khronos Group
• Open, multiple-vendor standard
• Support over a wide range of devices
  – GPUs
  – CPUs
  – Digital signal processors (DSPs)
  – Field-programmable gate arrays (FPGAs)
• Device kernels written in C
• Executing threads share the same kernel
• CPU-side code
  – C/C++
  – Very detailed CPU-side application programming interface (API)
  – Third-party bindings for Java, Python, etc.
GPU Applications

• Three components
  – Application algorithms
  – GPU kernel code
    • Can have multiple kernels per application
    • Each kernel usually contains an algorithm or algorithm step
    • Kernel code often uses GPU-specific techniques
  – CPU-side code
    • Moves OpenCL kernel to GPU
    • Manages GPU execution and errors
    • Moves application data between CPU and GPU
    • May contain a portion of application’s algorithms

• **DEFG’s domain is the CPU-side**
GPU Performance

• Major GPU Performance Concerns
  – Kernel Instruction Path Divergence
    • Due to conditional statements (ifs, loops, etc.)
    • Threads may pause
    • Minimize, if not totally avoid
  – High Memory Latency
    • One RAM access time equals time of 200-500 instructions
    • Accesses to global RAM should be coalesced

• Farber’s GPU suggestions [Farber2011]:
  – “Get the data on the GPU and leave it”
  – “Give the GPU enough work to do”
  – “Focus on data reuse to avoid memory limits”
DEFG Overview

• GPU software development tool for OpenCL
• Contains a Domain Specific Language (DSL)
  – Specialized computer language, focused on a domain
  – Developer writes CPU code with DEFG’s DSL
• Relative to hand-written code
  – *Faster development* by using declarative approach
  – *Simpler* by using design patterns and abstractions
• DEFG generates the corresponding CPU program
• Developer provides standard OpenCL GPU kernels
The DEFG generates C/C++ code for the CPU

DEFG Translator

- DEFG Source Input
- ANTLR-based Parser
- XML-based Tree
- Optimizer (Java)
- Code Generator (C++)
- Template Driven
- C/C++ Output

[Image of DEFG Translator Architecture]

[Refs: Senser2014]
DEFG Code Sample

01. declare application sobel
02. declare integer Xdim (0)
03. declare integer Ydim (0)
04. declare integer BUF_SIZE (0)
05. declaregpu gpusone (any)
06. declare kernel sobel_filter SobelFilter_Kernels ( [[ 2D,Xdim,Ydim ]] )
07. declare integer buffer image1 ( BUF_SIZE )
08. integer buffer image2 ( BUF_SIZE )
09. call init_input (image1(in) Xdim (out) Ydim (out) BUF_SIZE(out))
10. execute run1 sobel_filter ( image1(in) image2(out) )
11. call disp_output (image2(in) Xdim (in) Ydim (in) )
12. end

(Generates 440 lines of C/C++)

... status = clSetKernelArg(sobel_filter, 1, sizeof(cl_mem), (void *)&buffer_image2);
if (status != CL_SUCCESS) { handle error }
// *** execution
size_t global_work_size[2]; global_work_size[0] = Xdim ; global_work_size[1] = Ydim ;
status = clEnqueueNDRangeKernel(commandQueue, sobel_filter, 2, NULL, global_work_size, NULL, 0,
    NULL, NULL);
if (status != CL_SUCCESS) { handle error }
// *** result buffers
status = clEnqueueReadBuffer(commandQueue, buffer_image2, CL_TRUE, 0, BUF_SIZE * sizeof(int), image2, 0,
    NULL, NULL);
...
DEFG Benefits and Features

- Implements OpenCL applications with less effort
- Requires writing many fewer lines of code
- Encourages the developer to focus on the kernels
- How is this done?
  - With the Domain-Specific Language
    - Data characteristics are declared
    - Pre-defined DEFG design patterns are specified
    - Many implementation details are managed inside DEFG
  - Technical Features
    - Abstracts the OpenCL APIs, and their many details
    - Automatic optimization of buffer transfers
    - Supports multiple GPU devices
    - Handles error detection
DEFG Design Patterns

• Invocation Patterns (*Control Flow*)
  – Sequential-Flow
  – Single-Kernel Repeat Sequence
  – Multiple-Kernel

• Concurrent-GPU Patterns (*Multiple-GPU Support*)
  – Multiple-Execution
  – Divide-Process-Merge
  – Overlapped-Split-Process-Concatenate

• Other patterns include:
  – Prefix-Allocation (buffer allocation)
  – Code-Morsel (code insertion)
  – Anytime algorithm (control flow change on event)
  – BLAS-Usage (interface to Basic Linear Algebra Subprograms)

• Design patterns can be combined
  – Example: Sequential-Flow + Multiple-Execution + Divide-Process-Merge
Diverse DEFG Applications

• Demonstrate DEFG’s applicability
• Four diverse GPU application areas
  – *Image Filters*
    • Sobel Operator
    • Median Filter
    • Showcase for multiple GPU support
  – *Graph Theoretic*
    • Breadth-First Search with large graphs
    • Prefix-sum based buffer management
  – *Sorting*
    • Sorting partially sorted data
    • Prefix scan
  – *Numerical*
    • Iterative Matrix Inversion
    • clMath BLAS (Basic Linear Algebra Subprograms)
    • Anytime algorithm
Filter Application: Sobel Image Filter

• Sobel operator detected edges in images
• Pixel gradient was calculated from 3x3 mask
• A single GPU kernel was invoked once
• Example of DEFG Sobel operator processing:

Common uses: Object recognition, autonomous vehicle navigation, etc.
Filter Application: Median Filter

- Median determined for 5x5 mask
- Value at center of mask replaced by median value
- Like Sobel: a single GPU kernel, invoked once
- Example of DEFG median 5x5 filter processing:

Common uses: Electronic signal smoothing, noise removal, image preprocessing, etc.
**Application: Breadth-First Search (BFS)**

- Well-studied graph-theoretic problem
- Focus: BFS with Large Very Irregular (LVI) Graphs
  - Social Networks, Network Routing, A.I., etc.
- Numerous published GPU BFS approaches, starting with Harish [Harish2007]

- Harish used “Dijkstra” BFS
- Level-synchronous
- A GPU thread assigned to each vertex
- Vertex frontier stored as a Boolean array
List-Based BFS Vertex Frontier

• Merrill approach to vertex buffer management [Merrill2010]
  – Issue: list with multiple update threads
  – Solution: prefix sum to allocate buffer elements

• Shared buffers with multiple GPU devices
Goal: Improve on $O(n \log n)$ sorting bound when sequence is partially sorted

Based on the prior sorting work by T. Altman, et al. [Altman1989]

$k$ is a measure of “sortedness”

A sequence is $k$-sorted if no element is more than $k$ positions out of sequence

This $k$-sorted trait can be exploited

Knowing $k$ allows for sorts of $O(n \log k)$

If $k$ is small, obtain a substantial performance gain
Parallel Roughly Sorting Algorithm

Notion: Convert the large sort operation into many smaller, parallel sort operations.

Algorithm steps:

- LR: Left-to-right prefix scan (maximum)
- RL: Right-to-left prefix scan (minimum)
- DM: Computed distance measure using LR and RL
- UB: Computed upper bound of distance measure
  - This value became the $k$ value
  - Value used to determine size of sort blocks
- Sort: Individual blocks sorted in parallel
Iterative Matrix Inversion (IMI)

• Matrix inversion using M. Altman’s method [Altman1960]

• Required GPU matrix operations
  – Used OpenCL clMath BLAS library
  – Required clMath integration into DEFG

• With *anytime* approach
  – Inversion can produce early results
  – Balance run time against accuracy
  – Anytime management in DEFG
M. Altman IMI Approach

The initial inverse approximation, that is $R_0$, can be formed by:

$$R_0 = \alpha I$$

where $\alpha = 1 / \| A \|$

$\| A \|$ is the Euclidean norm of $A$

and $I$ is the identity matrix.

To invert matrix $A$, each iteration calculates:

$$R_{n+1} = R_n(3I - 3AR_n + (AR_n)^2).$$

- Better $R_0$ estimate provides for quicker convergence
- Application will end iterations when
  - Inversion quality measure is met
  - Maximum iterations have occurred
  - Anytime algorithm run-time limit is crossed

Example performance: $7,000 \times 7,000$ matrix inversion in 9 iterations
Accomplishments
DEFG Framework

• Fully Implemented
  – Consists of approximately 5,000 lines of code
  – 7 different applications
  – Complete User’s Guide
  – Packaged for general use

• Design Patterns
  – 12+ Patterns
  – Patterns designed to be combined

• Description of DEFG Limits
DEFG Usability and Performance

• Published DEFG Papers
  – Conference: Parallel and Distributed Processing Techniques and Applications (PDPTA’13) [Senser2013]
  – Conference: Parallel and Distributed Processing Techniques and Applications (PDPTA’14) [Senser2014]

• Existing OpenCL applications converted to DEFG
  1. Breadth-First Search (BFS)
  2. Floyd-Warshall (FW, All-Pairs Shortest Path)
  3. Sobel Image Filter (SOBEL)

• CPU-side re-coded in DEFG, used existing GPU kernels

• Comparisons between DEFG and existing applications
  – Lines-of-Code
  – Run-time Performance
On average, the DEFG code is 1/20th of the reference code size
• Shown are average run times
• CPU-based BFS-4096 was likely faster due to CPU’s cache

**Summary:** *DEFG provided equal, or better, performance*
Performance of Diverse Applications

• Implementations
  – Filtering
  – BFS
  – Sorting
  – Iterative Inversion

• Implementation Goals
  – Show general applicability of DEFG
  – Multiple-GPU: Filtering, BFS, and Sorting
  – Interesting Algorithms: Sorting and Iterative Inversion
  – BLAS Proof of Concept: Iterative Inversion

• Performance results
  – Problem-size characteristics
  – Run-time metrics
  – Observations for both single-GPU and multiple-GPU modes
  – Platform: C.S.E. Department’s Hydra server
Image Filtering

• Filtering Applications
  – Design patterns used in both SOBEL and MEDIAN
    • *Sequential-Flow*
    • *Multiple-Execution*
    • *Overlapped-Split-Process-Concatenate*

• Image Neighborhoods
  – SOBEL Application: 3x3 grid
  – MEDIAN Application: 5x5 grid

• SOBEL application refactored for multi-GPU use
  – Based upon earlier DEFG SOBEL application
  – Utilized existing OpenCL kernel
SOBEL Application

• Performance Tests
  – 50% image plus overlapped area given to each GPU
  – Produced identical image as 1-GPU version

• Run-time Performance with 2 GPUs
  – Run time was not as expected
    • OpenCL data transfer times went up
    • Kernel execution times stayed the same
  – Issue: computational workload not sufficiently intense
MEDIAN Application

• CPU-side DEFG code very similar to SOBEL
  – Developed OpenCL kernel for MEDIAN
  – More computationally intense

• Performance with 2-GPU, 5x5 MEDIAN
  – Run-time improvement with all test images
    • Example: Speedup of 1.34 (1.062 s / 0.794 s) with 7k by 7k image
    • Handled larger images (22k by 22k) than 1-GPU

• Performance Analysis with 2 GPUs
  – Kernel execution times dropped
  – OpenCL data transfer times increased
Breadth-First Search

- BFSDP2GPU Application Summary
  - Design patterns used in BFSDP2GPU
    - Multiple-Kernel
    - Multiple-Execution
    - Divide-Process-Merge
    - Prefix-Allocation
  - DEFG use of Merrill approach
    - Prefix-scan based buffer allocation
    - “Virtual pointers” to vertices
    - Shared buffers are dense data structures
    - Otherwise, kept Harish’s sparse data structures
Multiple-GPU BFS Implementation

- BFSDP2GPU DEFG Application
  - Based on earlier DEFG BFS application
    - Two kernels increased to six
    - Used two GPUs
  - Complex OpenCL application
    - Management of shared buffers
    - Run-time communications between GPUs

- Tested against LVI graphs
  - Test graphs from SNAP and DIMACS repositories
    - Stanford Network Analysis Package (SNAP) [SNAP2014]
    - Center for Discrete Mathematic and Theoretical Computer Science [DIMACS2010]
  - Very large graph datasets: millions of vertices and edges
BFSDP2GPU Performance Results

- Compared against existing DEFG BFS
- Processed large graphs (4.8M vertices, 69M edges)
- Run-time performance was not impressive
  - Run times increased by factors of 6 to 17
  - Issue: OpenCL's lack of GPU-to-GPU communications (77% of run-time, 0.59 of 0.771 seconds)
  - Lesser issue: mix of sparse and dense data structures
- External Experiment
  - Transfer rate comparison CUDA vs. OpenCL
  - CUDA GPU-to-GPU transfer: 21 times OpenCL rate
Roughly Sorting

• Design Patterns used in RSORT application
  – Multiple-Kernel
  – Multiple-Execution
  – Divide-Process-Merge

• GPU sort used: Comb Sort
  – sort-in-place design
  – non-recursive
  – similar to Bubble Sort, but much faster
  – elements are compared gap apart

• Five kernels: LRmax, RLmin, DM, UB, and comb_sort
RSORT Performance

• Comparison over three configurations
  – QSORT on CPU, fast sort used as baseline
  – RSORT with one GPU
  – RSORTM with two GPUs

• Run-time comparisons
  – Generated datasets with set $k$ values
  – Fully perturbed data
• Roughly Sorting’s run times impressive when \( k \) is small
• At \( K:2000 \), with \( 2^{26} \) items
  • Two-GPU RSORTM is faster than QSORT
  • Two-GPU versus One-GPU speedup near 2 (15.36 s/ 7.4 s)
• Second GPU adds sorting capacity
Iterative Matrix Inversion

• Design patterns used in IMIFLX application
  – Multiple-Kernel
  – BLAS-Usage

• Application characteristics
  – Blend of blas statements and kernels
    • blas for matrix multiplication
    • kernels for simpler matrix operations
  – Multiple blas statements per iteration
  – Anytime operation stopped iterating at time limit

• Analysis of application
  – Range of matrices: size and type
  – Inversion iterations
  – Data from University of Florida Sparse Matrix Collection
    [UFL2011]
IMIFLX Sample Result

- **Kuu Matrix**: 7,102 by 7,102 elements, sparse
- **Structural problem with 340,200 non-zero values**
- **9 iterations**
- **Norm value**: $\|/(A*R_n) - I\|$
Sample IMIIFLX Inversion Results

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size</th>
<th>Iterations</th>
<th>Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>H2</td>
<td>Hilbert</td>
<td>2x2</td>
<td>4</td>
<td>0.018</td>
</tr>
<tr>
<td>H12</td>
<td>Hilbert</td>
<td>12x12</td>
<td>70</td>
<td>0.089</td>
</tr>
<tr>
<td>M500</td>
<td>Generated</td>
<td>500x500</td>
<td>13</td>
<td>0.259</td>
</tr>
<tr>
<td>M8000</td>
<td>Generated</td>
<td>8000x8000</td>
<td>17</td>
<td>1380.320</td>
</tr>
<tr>
<td>1138_bus</td>
<td>Repository</td>
<td>1138x1138</td>
<td>14</td>
<td>3.262</td>
</tr>
<tr>
<td>Kuu</td>
<td>Repository</td>
<td>7102x7102</td>
<td>9</td>
<td>605.310</td>
</tr>
</tbody>
</table>

**Hydra’s NVIDIA T20 GPU**

- Available RAM: 2.68 GB
- Limits double-precision matrix size to just over 8,000 by 8,000
DEFG Generalization

• HPC with GPUs
  – Note the Faber suggestions for GPU performance:
    • “Get the data on the GPU and leave it”
    • “Give the GPU enough work to do”
    • “Focus on data reuse to avoid memory limits”
  – The CPU becomes the *orchestrator*

• DEFG provides the CPU code to orchestrate
  – Declarations to describe the data
  – Design patterns to describe the orchestration
  – Optimization to minimize the data transfers
Dissertation Accomplishments

• Designed, Implemented, and Tested DEFG

• Created DEFG’s Design Patterns

• Compared DEFG to Hand-Written Applications
  – DEFG required less code
  – DEFG produced equal or better run times

• Applied DEFG to Diverse GPU Applications
  – Each application fully implemented
  – Good application results
Future Research

• Additional DEFG Design Patterns
  – Multiple-GPU load balancing
  – Resource sharing

• GPU-side declarative approach

• DEFG Enhancements
  – Internal DSL, in addition to existing external DSL
    • More-standard programming environment
    • Enable support of more environments
  – Technical improvements
    • Better CPU RAM management
    • Additional collection of run-time statistics

• DEFG Support for NVIDIA’s CUDA
References


Additional Slides
DEFG Implementation Metrics

• Lines of Code
  – ANTLR-based parser:  580 lines
  – Optimizer:           660 lines of Java
  – Code Generator:      1,500 lines of C++
  – Templates and includes:  1,500 lines of C++

• Testing investment: 20% of total effort
  – Issues tended to be in the C/C++ code generation
  – Most were in multi-GPU buffer management
## Raw Performance Numbers for Three Applications, in Milliseconds

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU-Tesla T20</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DEFG</td>
<td>Ref.</td>
</tr>
<tr>
<td>BFS-4096</td>
<td>1.5</td>
<td>2.6</td>
</tr>
<tr>
<td>BFS-65536</td>
<td>12.3</td>
<td>14.2</td>
</tr>
<tr>
<td>FW</td>
<td>111.8</td>
<td>152.0</td>
</tr>
<tr>
<td>SOBEL</td>
<td>23.0</td>
<td>24.8</td>
</tr>
</tbody>
</table>
Sample DEFG Code Showing a Sequence

01. declare application floydwarshall
02. declare integer NODE_CNT (0)
03. declare integer BUF_SIZE (0)
04. declare gpu gpuone ( any )
05. declare kernel floydWarshallPass FloydWarshall_Kernels ( [[ 2D,NODE_CNT ]])
06. declare integer buffer buffer1 ( BUF_SIZE )
07. integer buffer buffer2 ( BUF_SIZE )
08. call init_input (buffer1(in) buffer2(in) NODE_CNT(out) BUF_SIZE(out))
09. sequence NODE_CNT times
10. execute run1 floydWarshallPass ( buffer1(inout) buffer2(out) NODE_CNT(in) DEFG_CNT(in) )
11. call disp_output (buffer1(in) buffer2(in) NODE_CNT(in))
12. end
declare application bfs
declare integer NODE_CNT (0)
declare integer EDGE_CNT (0)
declare integer STOP (0)
declare gpu gpuone ( any )
declare kernel kernel1 bfs_kernel ( [[ 1D,NODE_CNT ]] )
kernel kernel2 bfs_kernel ( [[ 1D,NODE_CNT ]] )
declare struct (4) buffer graph_nodes ( NODE_CNT )
    integer buffer graph_edges ( EDGE_CNT )
    integer buffer graph_mask ( NODE_CNT )
    integer buffer updating_graph_mask ( $NODE_CNT )
    integer buffer graph_visited ( NODE_CNT )
    integer buffer cost ( NODE_CNT )
// note: init_input handles setting "source" node
call init_input (graph_nodes(out) graph_edges(out) graph_mask(out) updating_graph_mask(out) graph_visited (out) cost (out) NODE_CNT(out) EDGE_CNT(out))

loop
execute part1 kernel1 ( graph_nodes(in) 
    graph_edges(in)
    graph_mask(in)
    updating_graph_mask(out)
    graph_visited(in)
    cost(inout)
    $NODE_CNT(in) )
// set STOP to zero each time thru...
set STOP (0)
// note: STOP value is returned...
execute part2 kernel2 ( graph_mask(inout) 
    updating_graph_mask(inout)
    graph_visited(inout)
    STOP(inout)
    NODE_CNT(in) )
while STOP eq 1
    call disp_output (cost(in) NODE_CNT(in))
end
### Table 5.13: Sort Run Times on Hydra with $2^{26}$ Items, in Seconds

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Qsort</td>
<td>8.394</td>
<td>8.008</td>
<td>7.972</td>
<td>7.922</td>
<td>7.890</td>
</tr>
<tr>
<td>Rsort</td>
<td>2.527</td>
<td>11.216</td>
<td>15.360</td>
<td>17.120</td>
<td>29.556</td>
</tr>
<tr>
<td>RsortM</td>
<td>1.459</td>
<td>6.487</td>
<td>7.400</td>
<td>11.189</td>
<td>24.682</td>
</tr>
</tbody>
</table>

### Table 5.14: Sort Run Times on Hydra with $2^{27}$ Items, in Seconds

|--------------|-----------|-------------|-------------|-------------|-------------|
# IMIFLX Data

## Table 5.16: IMIFLX Inversion Results for Various Matrices

<table>
<thead>
<tr>
<th>Cnt</th>
<th>Matrix Name</th>
<th>Type</th>
<th>Size</th>
<th>Epilon</th>
<th>Iterations</th>
<th>Run Time Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>H2</td>
<td>Hilbert</td>
<td>2x2</td>
<td>0.000001</td>
<td>4</td>
<td>0.018</td>
</tr>
<tr>
<td>2</td>
<td>H3</td>
<td>Hilbert</td>
<td>3x3</td>
<td>0.000001</td>
<td>8</td>
<td>0.022</td>
</tr>
<tr>
<td>3</td>
<td>H4</td>
<td>Hilbert</td>
<td>4x4</td>
<td>0.000001</td>
<td>12</td>
<td>0.023</td>
</tr>
<tr>
<td>4</td>
<td>H5</td>
<td>Hilbert</td>
<td>5x5</td>
<td>0.000001</td>
<td>15</td>
<td>0.030</td>
</tr>
<tr>
<td>5</td>
<td>H6</td>
<td>Hilbert</td>
<td>6x6</td>
<td>0.000001</td>
<td>18</td>
<td>0.034</td>
</tr>
<tr>
<td>6</td>
<td>H7</td>
<td>Hilbert</td>
<td>7x7</td>
<td>0.000001</td>
<td>21</td>
<td>0.036</td>
</tr>
<tr>
<td>7</td>
<td>H8</td>
<td>Hilbert</td>
<td>8x8</td>
<td>0.000001</td>
<td>24</td>
<td>0.037</td>
</tr>
<tr>
<td>8</td>
<td>H9</td>
<td>Hilbert</td>
<td>9x9</td>
<td>0.000001</td>
<td>27</td>
<td>0.042</td>
</tr>
<tr>
<td>9</td>
<td>H10</td>
<td>Hilbert</td>
<td>10x10</td>
<td>0.001</td>
<td>30</td>
<td>0.035</td>
</tr>
<tr>
<td>10</td>
<td>H11</td>
<td>Hilbert</td>
<td>11x11</td>
<td>0.005</td>
<td>40</td>
<td>0.057</td>
</tr>
<tr>
<td>11</td>
<td>H12</td>
<td>Hilbert</td>
<td>12x12</td>
<td>0.15</td>
<td>70</td>
<td>0.089</td>
</tr>
<tr>
<td>12</td>
<td>H13</td>
<td>Hilbert</td>
<td>13x13</td>
<td>n.a.</td>
<td>n.a.</td>
<td>#INF error</td>
</tr>
<tr>
<td>13</td>
<td>M500</td>
<td>Invertible</td>
<td>500x500</td>
<td>0.000001</td>
<td>13</td>
<td>0.259</td>
</tr>
<tr>
<td>13a</td>
<td>M500</td>
<td>Invt-AnyTime</td>
<td>500x500</td>
<td>0.000001</td>
<td>10</td>
<td>0.206</td>
</tr>
<tr>
<td>14</td>
<td>M1000</td>
<td>Invertible</td>
<td>1000x1000</td>
<td>0.000001</td>
<td>14</td>
<td>2.112</td>
</tr>
<tr>
<td>15</td>
<td>M5000</td>
<td>Invertible</td>
<td>5000x5000</td>
<td>0.000001</td>
<td>16</td>
<td>329.619</td>
</tr>
<tr>
<td>16</td>
<td>M8000</td>
<td>Invertible</td>
<td>8000x8000</td>
<td>0.000001</td>
<td>17</td>
<td>1380.320</td>
</tr>
<tr>
<td>17</td>
<td>M8500</td>
<td>Invertible</td>
<td>8500x8500</td>
<td>n.a.</td>
<td>n.a.</td>
<td>error -4</td>
</tr>
<tr>
<td>18</td>
<td>685_bus</td>
<td>Repository</td>
<td>685x685</td>
<td>0.000001</td>
<td>12</td>
<td>0.665</td>
</tr>
<tr>
<td>19</td>
<td>1138_bus</td>
<td>Repository</td>
<td>1138x1138</td>
<td>0.000001</td>
<td>14</td>
<td>3.262</td>
</tr>
<tr>
<td>20</td>
<td>Kuu</td>
<td>Repository</td>
<td>7102x7102</td>
<td>0.000001</td>
<td>9</td>
<td>605.310</td>
</tr>
</tbody>
</table>
# DEFG 4-Way Mini-Experiment SpeedUp

<table>
<thead>
<tr>
<th>GPUs</th>
<th>SpeedUp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1.947</td>
</tr>
<tr>
<td>4</td>
<td>3.622</td>
</tr>
</tbody>
</table>

![Graph of 4-GPU SpeedUp](image)