Accelerating Financial Applications on the GPU

Scott Grauer-Gray    William Killian
Robert Searles      John Cavazos

Department of Computer and Information Science
University of Delaware

Sixth Workshop on General Purpose Processing Using GPUs
Outline

1. Introduction
   - QuantLib and Financial Applications
   - Directive-Based Acceleration
2. Experiment Setup
   - Source Code Modifications
   - Compilation
   - Execution Environment
3. Application Results
   - NVIDIA K20 Results
4. Auto-Tuning
   - Framework
   - Results
   - Alternate Architectures
5. Conclusion
   - Future Work
   - Final Notes
QuantLib

- Open-Source library for Quantitative Finance
- Written in C++
- Contains various financial models and methods
  - Models: yield curves, interest rates, volatility
  - Methods: analytic formulae, finite difference, monte-carlo
- Financial applications optimized are particular code paths in QuantLib
### Financial Applications

Four financial applications selected for parallelization

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black-Scholes</td>
<td>Option pricing using Black-Scholes-Merton pricing</td>
<td>Single</td>
</tr>
<tr>
<td>Monte-Carlo</td>
<td>Pricing of a single option using QMB (Sobol) Monte-Carlo method</td>
<td>Single</td>
</tr>
<tr>
<td>Bonds</td>
<td>Bond pricing using a fixed-rate bond with a flat forward-curve</td>
<td>Double</td>
</tr>
<tr>
<td>Repo</td>
<td>Repurchase agreement pricing of securities which are sold and bought back later</td>
<td>Double</td>
</tr>
</tbody>
</table>

- Each application is data-parallelized
- Algorithm for each application is parallelized where possible
Overview on Directive-Based Acceleration

- Syntax comparable to OpenMP
- Annotates what code should run on an accelerator
- Focuses on highlighting parallelism of code
- Preserves serial implementation of code
- Simplifies interaction between scientists and programmers
Directive-Based Programming Languages

OpenACC
- Joint collaboration between CAPS Entreprise, CRAY, PGI, and NVIDIA
- Directive syntax near identical to OpenMP with added data clauses
- Introduces a kernel directive that drives compiler-assisted parallelization

HMPP
- Originally developed by CAPS Entreprise
- Fundamental execution unit is a codelet
- Provides fine-grain control for optimizations
Source Code Modifications

- Code flatten QuantLib C++ ⇒ Sequential C code
- Implementations derived from Sequential C code
- Argument passing — Structure of Arrays
- **Verification**: Compared all results to original QuantLib code paths. All results were within 3 degrees of precision ($10^{-3}$)
// C++ code:
struct C {
    int x;
    void addFour() {
        x += 4;
    }
};
struct B {
    C myObj;
    virtual void foo() = 0;
};
struct A : public B {
    virtual void foo() {
        myObj.addFour();
    }
};
A inst;
inst.foo();

// flattened code:
int inst_x;
inst_x += 4;

// Alternative flattening:
int addFour (int x) {
    return x + 4;
}
int inst_x;
inst_x = addFour (inst_x);
Compilation

- Host code compiled with GCC 4.7.0
  - `-O2` flag used for serial
  - `-O3 -march=native` flag used for OpenMP
- OpenACC and HMPP compiled with HMPP Workbench 3.2.1
- CUDA compiled with CUDA 5 Toolkit
- OpenCL used NVIDIA driver version 304.54
Compile Workflow Using HMPP Workbench

- HMPP Workbench used for HMPP and OpenACC code compilation
- Target CUDA and OpenCL code generation
Execution Environment

**CPU** — Dual Xeon X5530 (Quad-Core @ 2.40GHz) with 24GB DDR3-1066 ECC RAM

**GPU** — NVIDIA K20c (2496 CUDA Cores @ 706MHz) with 5GB GDDR5 2.6GHz ECC RAM

**NOTE:** Also ran all experiments on NVIDIA C2050

**Auto-Tuning Targets:**

<table>
<thead>
<tr>
<th>NVIDIA GPU</th>
<th>Architecture</th>
<th>CUDA Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA C1060</td>
<td>Tesla</td>
<td>240</td>
</tr>
<tr>
<td>NVIDIA C2050</td>
<td>Fermi</td>
<td>448</td>
</tr>
<tr>
<td>NVIDIA GTX 670</td>
<td>Kepler GK104</td>
<td>1344</td>
</tr>
<tr>
<td>NVIDIA K20c</td>
<td>Kepler GK110</td>
<td>2496</td>
</tr>
</tbody>
</table>
Outline

1. Introduction
   - QuantLib and Financial Applications
   - Directive-Based Acceleration

2. Experiment Setup
   - Source Code Modifications
   - Compilation
   - Execution Environment

3. Application Results
   - NVIDIA K20 Results

4. Auto-Tuning
   - Framework
   - Results
   - Alternate Architectures

5. Conclusion
   - Future Work
   - Final Notes
Black-Scholes — K20 Results

CUDA Results

OpenCL Results

Number of Options

Speedup over Sequential

OpenACC
HMPP
CUDA
OpenMP

OpenACC
HMPP
OpenCL
OpenMP

100
200
500
1000
2000
5000
10000
20000
50000
100000
200000
500000
1000000
2000000
5000000
10
0
10
1
10
2

Number of Options

Speedup over Sequential

100
200
500
1000
2000
5000
10000
20000
50000
100000
200000
500000
1000000
2000000
5000000
10
0
10
1
10
2
Black-Scholes — K20 Results

• CUDA outperformed OpenCL on NVIDIA K20
  • 461x speedup for CUDA
  • 446x speedup for OpenCL
• HMPP and OpenACC targeting the same language achieved near-identical speedup
• HMPP and OpenACC targeting OpenCL was faster than targeting CUDA
  • 369x speedup for CUDA
  • 380x speedup for OpenCL
Random Number Generation:
- C/OpenMP — rand
- CUDA — cuRand
- HMPP/OpenACC/OpenCL — Mersenne Twister

Dropoff in speedup for CUDA ⇒ cache misses
Monte-Carlo — K20 Results

- Manual CUDA outperformed manual OpenCL
  - Up to 1006x vs 180x
- HMPP and OpenACC performed similarly
- Targeting CUDA was faster than targeting OpenCL
  - Up to 162x vs up to 130x
Bonds and Repo — K20 Results

Problem: Generating OpenCL code from HMPP and OpenACC
Bonds and Repo — K20 Results

- Bonds: Up to 87.9x speedup
- Repo: Up to 94x speedup
- HMPP and OpenACC versions produced near-identical execution time
- HMPP and OpenACC versions ran within 2% execution time as manually-written CUDA
- Speedup flattened as problem size increased beyond 100,000 Bonds and 2,000,000 Repos
Outline

1. Introduction
   - QuantLib and Financial Applications
   - Directive-Based Acceleration

2. Experiment Setup
   - Source Code Modifications
   - Compilation
   - Execution Environment

3. Application Results
   - NVIDIA K20 Results

4. Auto-Tuning
   - Framework
   - Results
   - Alternate Architectures

5. Conclusion
   - Future Work
   - Final Notes
Goal: achieve maximum speedup by applying a set optimizations (while preserving accuracy)

Collection of python scripts initially provided by CAPS Entreprise

Injects code optimizations into annotated source code
  - blocksize — thread block dimensions on GPU
  - unroll — loop unroll factor; can be used with contiguous or split
  - tile — loop tiling factor
  - remainder/guarded — used for unrolling to specify remainder loop or conditional check, respectively

Framework generates a set of new HMPP source files
Annotated Source Code Sample

```c
%(blocksizePragma)
%(unrollTilePragma_iLoop)
%(parallelNoParallelPragma_iLoop)
for (i = 0; i < NI; ++i) {
  %(unrollTilePragma_jLoop)
  %(parallelNoParallelPragma_jLoop)
  for (j = 0; j < NJ; ++j) {
    c[i][j] *= p_beta;
    %(unrollTilePragma_kLoop)
    %(parallelNoParallelPragma_kLoop)
    for (k = 0; k < NK; ++k) {
      temp = p_alpha * a[i][k] * b[k][j];
      c[i][j] += temp;
    }
  }
}
```

- `unrollTilePragma` — specify loop unroll/tile factor with options
- `parallelNoParallelPragma` — specify whether to parallelize or not
- `blockSizePragma` — use determined block size
## Auto-Tuning Results

<table>
<thead>
<tr>
<th>Application</th>
<th>Thread Block</th>
<th>Loop Optimizations</th>
<th>Speedup (Default)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black-Scholes 5,000,000 Options</td>
<td>32 X 4</td>
<td>No tiling / loop unrolling</td>
<td>369x (369x)</td>
</tr>
<tr>
<td>Monte-Carlo 400,000 Samples</td>
<td>32 X 2</td>
<td>Tile ‘main’ loop w/ factor 3 and ‘path’ loop w/ factor 4, both with ‘contiguous’ and ‘guarded’ options</td>
<td>265x (152x)</td>
</tr>
<tr>
<td>Bonds 1,000,000 Bonds</td>
<td>32 X 2</td>
<td>No tiling / loop unrolling</td>
<td>89.7x (87.1x)</td>
</tr>
<tr>
<td>Repo 1,000,000 Repos</td>
<td>32 X 2</td>
<td>Unroll inner ‘cash flows’ loop w/ factor 2 using ‘split’ and ‘guarded’ options</td>
<td>97.6x (91.2x)</td>
</tr>
</tbody>
</table>
Running Optimized Code on Alternate Architectures

- Run the auto-tuned code on various architectures
- Compare speedup of best auto-tuned code of one architecture on other architecture
- All code paths executed on C1060, **C2050**, and **GTX670**

![Speedup over default for C2050](chart)

![Speedup over default for GTX670](chart)
Outline

1. Introduction
   - QuantLib and Financial Applications
   - Directive-Based Acceleration

2. Experiment Setup
   - Source Code Modifications
   - Compilation
   - Execution Environment

3. Application Results
   - NVIDIA K20 Results

4. Auto-Tuning
   - Framework
   - Results
   - Alternate Architectures

5. Conclusion
   - Future Work
   - Final Notes
Future Work

- Target different architectures
  - AMD GPUs
  - Intel Xeon Phi
  - Heterogeneous systems
- Parallelize more code paths in QuantLib
- Parallelize additional financial applications outside of QuantLib
Successful parallelization of four QuantLib code paths

- Achieve up to a 1000x speedup by targeting CUDA manually
- Achieve up to a 370x speedup by using HMPP and OpenACC
- Achieve up to a 74% speedup when auto-tuning

Source code for codes in this presentation will be available at [www.sourceforge.net/projects/quantlib-gpu/](http://www.sourceforge.net/projects/quantlib-gpu/)

Funding Acknowledgement:
This work was funded in part by JP Morgan Chase as part of the Global Enterprise Technology (GET) Collaboration