Dynamic Code Generation and Execution for Monte Carlo Simulations

Vaivaswatha Nagaraj
Steve Karmesin
Outline

- Introduction
- Code Generation
- Compilation & Execution
- Results
- Conclusion and Future Work
Outline

- Introduction
- Code Generation
- Compilation & Execution
- Results
- Conclusion and Future Work
Monte Carlo Simulation

- Numerical method to find probabilities of outcomes in a process
- Useful when closed-form solutions are absent (or difficult to find)
- Widely used in a variety of domains: physics, engineering, finance etc.
Monte Carlo Simulation

- Numerical method to find probabilities of outcomes in a process
- Useful when closed-form solutions are absent (or difficult to find)
- Widely used in a variety of domains: physics, engineering, finance etc.
- Inherently data-parallel: Computations over different paths are independent

\[ p = f(X_0, \ldots, X_i, C_0, \ldots, C_i) \]

- \( X_0 \ldots X_i \): random variables
- \( C_0, \ldots, C_i \): parameters or constants
Monte Carlo Simulation for Derivative Pricing

Script → Instrument Model → Pricing Engine → Sequence of Vector Operations (Computations for Monte-Carlo simulation) → Execute
Monte Carlo Vector Operation Sequence

v1 = {0.000138513}
v2 = {rand_normal()}
v3 = { ... }
v1 = {pow(v2, v1)}
v1 = v3 * v1
Monte Carlo Vector Operation Sequence

\[ v_1 = \{0.000138513\} \]
\[ v_2 = \{\text{rand}\_\text{normal}\()\} \]
\[ v_3 = \{\ldots\} \]
\[ v_1 = \{\text{pow}(v_2, v_1)\} \]
\[ v_1 = v_3 \times v_1 \]

for (i = 0; i < n; i++)
\[ v_1[i] = 0.000138513; \]
for (i = 0; i < n; i++)
\[ v_2[i] = \text{rand}\_\text{normal}(); \]
for (i = 0; i < n; i++)
\[ v_3[i] = \ldots \]
for (i = 0; i < n; i++)
\[ v_1[i] = \text{pow}(v_2[i], v_1[i]); \]
for (i = 0; i < n; i++)
\[ v_1[i] = v_3[i] \times v_1[i]; \]
Monte Carlo Vector Operation Sequence

\( v_1 = \{0.000138513\} \)
\( v_2 = \{\text{rand\_normal()}\} \)
\( v_3 = \{ \ldots \} \)
\( v_1 = \{\text{pow}(v_2, v_1)\} \)
\( v_1 = v_3 \times v_1 \)

for (i = 0; i < n; i++)
\( v_1[i] = 0.000138513; \)
for (i = 0; i < n; i++)
\( v_2[i] = \text{rand\_normal}(); \)
for (i = 0; i < n; i++)
\( v_3[i] = \ldots \)
for (i = 0; i < n; i++)
\( v_1[i] = \text{pow}(v_2[i], v_1[i]); \)
for (i = 0; i < n; i++)
\( v_1[i] = v_3[i] \times v_1[i]; \)
Loop Fusion for Locality

for (i = 0; i < n; i++)
    v1[i] = 0.000138513;
for (i = 0; i < n; i++)
    v2[i] = rand_normal();
for (i = 0; i < n; i++)
    v3[i] = …
for (i = 0; i < n; i++)
    v1[i] = pow(v2[i], v1[i])
for (i = 0; i < n; i++)
    v1[i] = v3[i] * v1[i];

No temporal locality

for (i = 0; i < n; i++) {
    t1 = 0.000138513;
    t2 = rand_normal();
    t3 = …;
    t1 = pow(t2, t1);
    v1[i] = t3 * t1;
}

Temporal locality / fewer memory accesses

Loop Fusion
for (i = 0; i < n; i++)
    v1[i] = 0.000138513;
for (i = 0; i < n; i++)
    v2[i] = rand_normal();
for (i = 0; i < n; i++)
    v3[i] = …
for (i = 0; i < n; i++)
    v1[i] = pow(v2[i], v1[i]);
for (i = 0; i < n; i++)
    v1[i] = v3[i] * v1[i];

We do not know the sequence of operations until execution. Cannot do loop-fusion.

Solution: generate this loop on-the-fly and execute it.
Advantages

- Preserves existing APIs and workflow
  - Clients include hundreds of financial companies
  - Software is millions of lines of code large
- The advantage of JIT compilation
  - Better code optimization
Outline

- Introduction
- Code Generation
- Compilation & Execution
- Results
- Conclusion and Future Work
PTX Representation

- In-house PTX generator
  - Minimal
  - Fast
- Emits text PTX
- Significantly faster than LLVM PTX backend
Kernel Re-use

- Full pricings involve multiple executions of a function, with different parameters / literal constants
- Parameters are not hard-coded, but loaded from constant bank
- Low over-head
- Re-use across different pricing runs

\[ p = f(X_0, \ldots, X_i, C_0, \ldots, C_i) \]

\( X_0 \ldots X_i \) : random variables
\( C_0, \ldots, C_i \) : parameters or constants
Outline

- Introduction
- Code Generation
- Compilation & Execution
- Results
- Conclusion and Future Work
JIT Compilation

- CUDA driver API for JIT compilation of generated PTX
- CUDA driver caches compiled kernels
- Small optimizations before calling the CUDA compiler
External/Library Functions

- External calls to math functions (log, exp. etc.,) and our own custom functions for specific operations
- Support for external functions
  - Library of PTX text definitions of external functions that can be called
  - Included with and JIT’ed along with main kernel code (relying on driver cache mechanism)
  - Disadvantage: Difficult to maintain
External/Library Functions

- PTXLib.cu
  - nvcc
  - PTXLib.ptx
  - Generated PTX
  - JIT compile/link
  - CUModule
  - Execute

- static
- dynamic
Outline

- Introduction
- Code Generation
- Compilation & Execution
- Results
- Conclusion and Future Work
System Configuration

- Quadro M1000M GPU on a laptop with Core i7-6820HQ @ 2.7 GHz CPU.
- Windows 10 Pro
- CUDA 8.0
- 16GB main memory and 2GB GPU memory
Benchmarks

1. Multi-equity option with knock-out barriers.
2. Hybrid model with three equities and a deterministic IR model.
3. Three equity option to compute “Greeks”.
4. Variable Annuity product.
100k Monte-Carlo Paths

Speedup using DCGE

- Knock-out Barrier: Speedup considering JIT overhead = 0.71, Speedup ignoring JIT overhead = 2.5
- Hybrid model: Speedup considering JIT overhead = 0.69, Speedup ignoring JIT overhead = 2.5
- Greek Computation: Speedup considering JIT overhead = 0.55, Speedup ignoring JIT overhead = 4.9, JIT overhead = 0.88
- Variable Annuity: Speedup considering JIT overhead = 1, Speedup ignoring JIT overhead = 1.9, JIT overhead = 0.45

JIT overhead (fraction of total time)
300k Monte-Carlo Paths

![Speedup using DCGE](image)

- **Knock-out Barrier**: Speedup considering JIT overhead: 0.5, Speedup ignoring JIT overhead: 1.5, JIT overhead: 0.22 (fraction of total time)
- **Hybrid model**: Speedup considering JIT overhead: 0.8, Speedup ignoring JIT overhead: 1.3, JIT overhead: 0.55 (fraction of total time)
- **Greek Computation**: Speedup considering JIT overhead: 0.76, Speedup ignoring JIT overhead: 3.5, JIT overhead: 0.5
- **Variable Annuity**: Speedup considering JIT overhead: 1.9, Speedup ignoring JIT overhead: 1.5, JIT overhead: 0.22 (fraction of total time)
500k Monte-Carlo Paths

Speedup using DCGE

- Knock-out Barrier: Speedup considering JIT overhead = 2.5, Speedup ignoring JIT overhead = 4.7, JIT overhead = 0.46
- Hybrid model: Speedup considering JIT overhead = 1.9, Speedup ignoring JIT overhead = 3.4, JIT overhead = 0.44
- Greek Computation: Speedup considering JIT overhead = 1.1, Speedup ignoring JIT overhead = 3.5, JIT overhead = 0.68

JIT overhead (fraction of total time)
Outline

- Introduction
- Code Generation
- Compilation & Execution
- Results
- Conclusion and Future Work
Conclusion and Future Work

- At least 2x speedup in most cases
- Explore using LLVM for PTX generation
- Use the technique for CPU execution also
Questions?

Contact
vnagaraj@numerix.com
karmesin@numerix.com

Thank you
Backup Slide 1 - Execution times

**Knockout Barrier**

<table>
<thead>
<tr>
<th>Number of Monte Carlo Paths</th>
<th>50000</th>
<th>100000</th>
<th>200000</th>
<th>300000</th>
<th>500000</th>
</tr>
</thead>
<tbody>
<tr>
<td>No DCGE</td>
<td>0.096</td>
<td>0.160</td>
<td>0.324</td>
<td>0.469</td>
<td>0.801</td>
</tr>
<tr>
<td>DCGE</td>
<td>0.211</td>
<td>0.224</td>
<td>0.239</td>
<td>0.294</td>
<td>0.314</td>
</tr>
<tr>
<td>JIT overhead (part of DCGE)</td>
<td>0.151</td>
<td>0.162</td>
<td>0.155</td>
<td>0.150</td>
<td>0.145</td>
</tr>
</tbody>
</table>

**Hybrid Model**

<table>
<thead>
<tr>
<th>Number of Monte Carlo Paths</th>
<th>50000</th>
<th>100000</th>
<th>200000</th>
<th>300000</th>
<th>500000</th>
</tr>
</thead>
<tbody>
<tr>
<td>No DCGE</td>
<td>5.94</td>
<td>9.9</td>
<td>20.0</td>
<td>29.3</td>
<td>49.5</td>
</tr>
<tr>
<td>DCGE</td>
<td>13.3</td>
<td>14.3</td>
<td>18.0</td>
<td>21.0</td>
<td>26.0</td>
</tr>
<tr>
<td>JIT overhead (part of DCGE)</td>
<td>10.6</td>
<td>10.4</td>
<td>11.1</td>
<td>11.7</td>
<td>11.6</td>
</tr>
</tbody>
</table>
## Backup Slide 2 - Execution times

### Greek Computation

<table>
<thead>
<tr>
<th>Number of Monte Carlo Paths</th>
<th>50000</th>
<th>100000</th>
<th>200000</th>
<th>300000</th>
<th>500000</th>
</tr>
</thead>
<tbody>
<tr>
<td>No DCGE</td>
<td>1.61</td>
<td>1.98</td>
<td>2.7</td>
<td>3.5</td>
<td>5.3</td>
</tr>
<tr>
<td>DCGE</td>
<td>3.4</td>
<td>3.6</td>
<td>3.8</td>
<td>4.2</td>
<td>4.7</td>
</tr>
<tr>
<td>JIT overhead (part of DCGE)</td>
<td>3.1</td>
<td>3.2</td>
<td>3.2</td>
<td>3.2</td>
<td>3.2</td>
</tr>
</tbody>
</table>

### Variable Annuity

<table>
<thead>
<tr>
<th>Number of Monte Carlo Paths</th>
<th>50000</th>
<th>100000</th>
<th>200000</th>
<th>300000</th>
<th>500000</th>
</tr>
</thead>
<tbody>
<tr>
<td>No DCGE</td>
<td>45.2</td>
<td>85.5</td>
<td>162.7</td>
<td>244.6</td>
<td>-</td>
</tr>
<tr>
<td>DCGE</td>
<td>62.3</td>
<td>82.0</td>
<td>121.7</td>
<td>162.9</td>
<td>242.2</td>
</tr>
<tr>
<td>JIT overhead (part of DCGE)</td>
<td>37.1</td>
<td>37.0</td>
<td>37.3</td>
<td>37.3</td>
<td>37.5</td>
</tr>
</tbody>
</table>