GPU TECHNOLOGY CONFERENCE

Dynamic Code Generation and Execution for Monte Carlo Simulations

Vaivaswatha Nagaraj

Steve Karmesin

numerix

Talk ID: 23282

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.

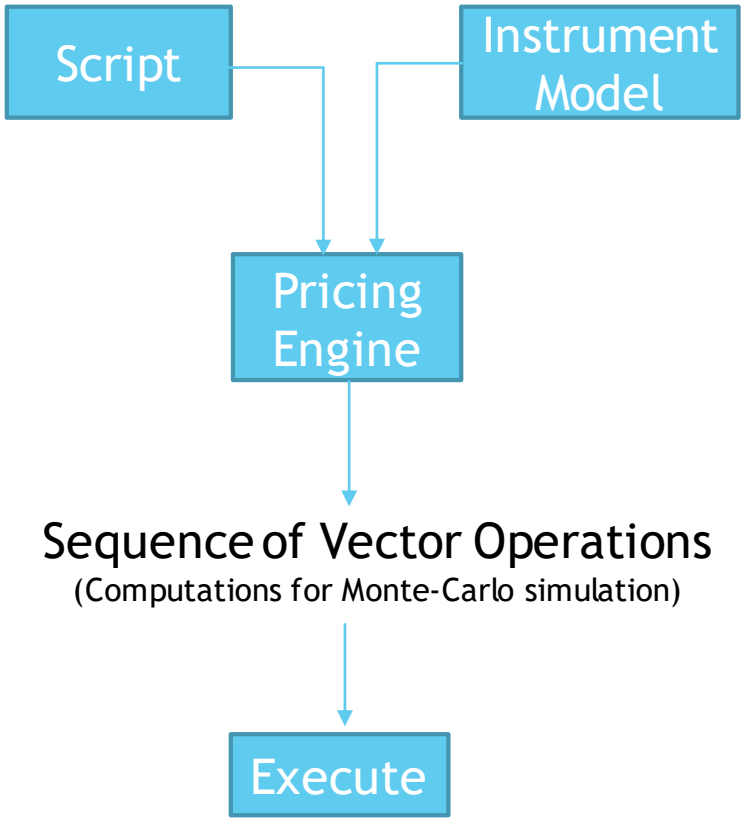
Monte Carlo Simulation

- 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

$$\mathrm{p} = f(X_0, \ldots, X_i, C_0, \ldots C_i)$$

$X_0 \ldots \mathrm{X_i}$: random variables
$C_0, \ldots C_i$: parameters or constants

Monte Carlo Simulation for Derivative Pricing

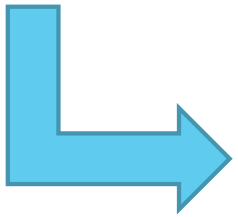


Monte Carlo Vector Operation Sequence

```
v1 = {0.000138513}
v2 = {rand_normal()}
v3 = { … }
v1 = {pow(v2, v1)}
v1 = v3 * v1
```

Monte Carlo Vector Operation Sequence

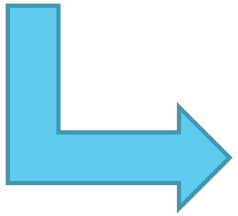
```
v1 = {0.000138513}
v2 = {rand_normal()}
v3 = { … }
v1 = {pow(v2, v1)}
v1 = v3 * v1
```



```
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];
```

Monte Carlo Vector Operation Sequence

```
v1 = {0.000138513}
v2 = {rand_normal()}
v3 = { … }
v1 = {pow(v2, v1)}
v1 = v3 * v1
```



```
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

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

Loop Fusion

Temporal locality / fewer memory accesses

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

Dynamic Code Generation and Execution

```
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.

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

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

$$\mathrm{p} = f(X_0, \ldots, X_i, C_0, \ldots C_i)$$

$X_0 \ldots \mathrm{X_i}$: random variables
$C_0, \ldots C_i$: parameters or constants

Outline

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

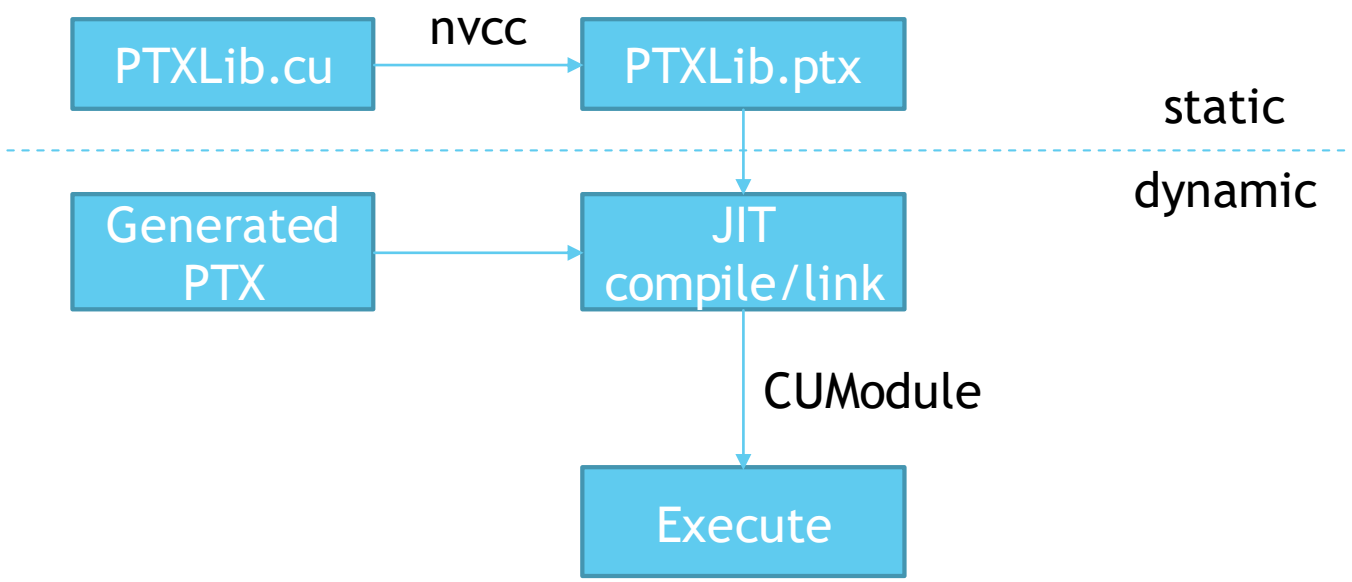
JIT Compilation

- 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



Outline

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

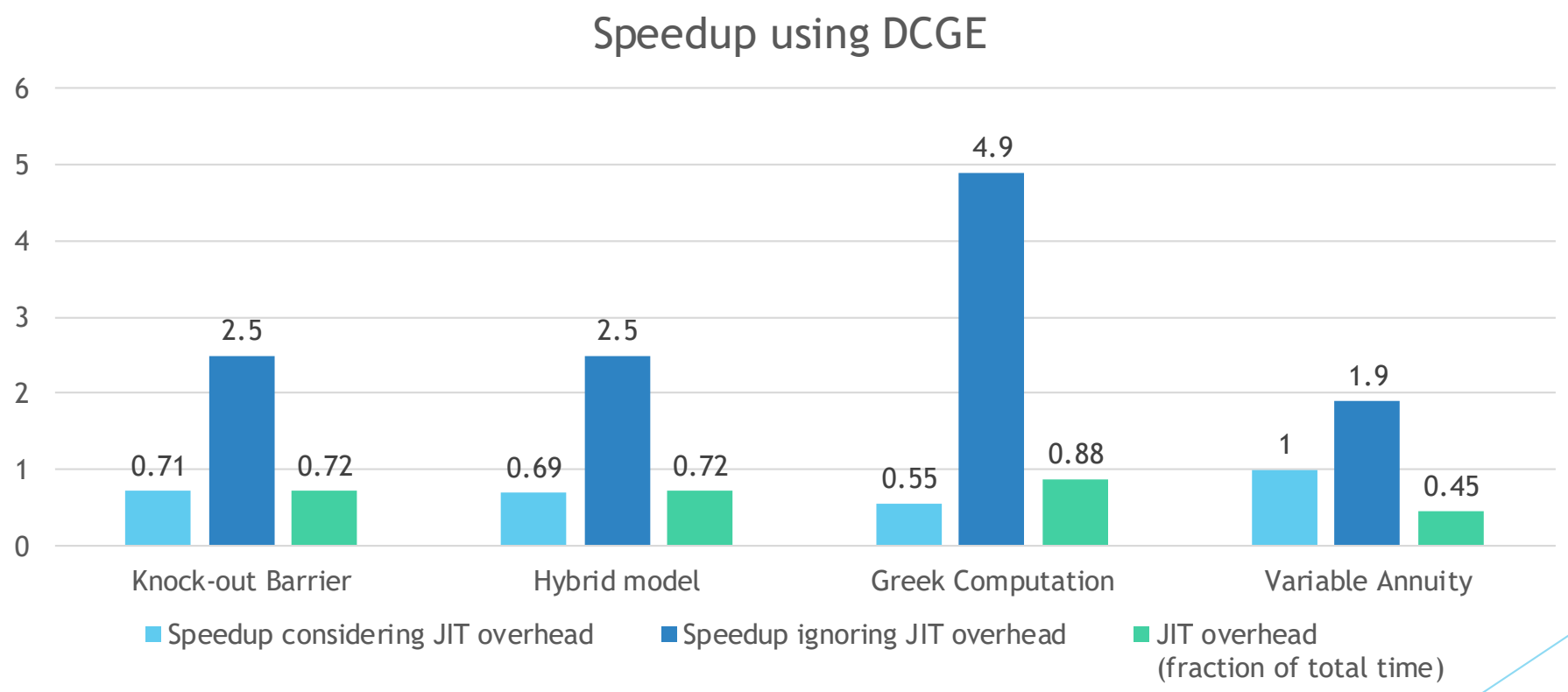
System Configuration

- 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

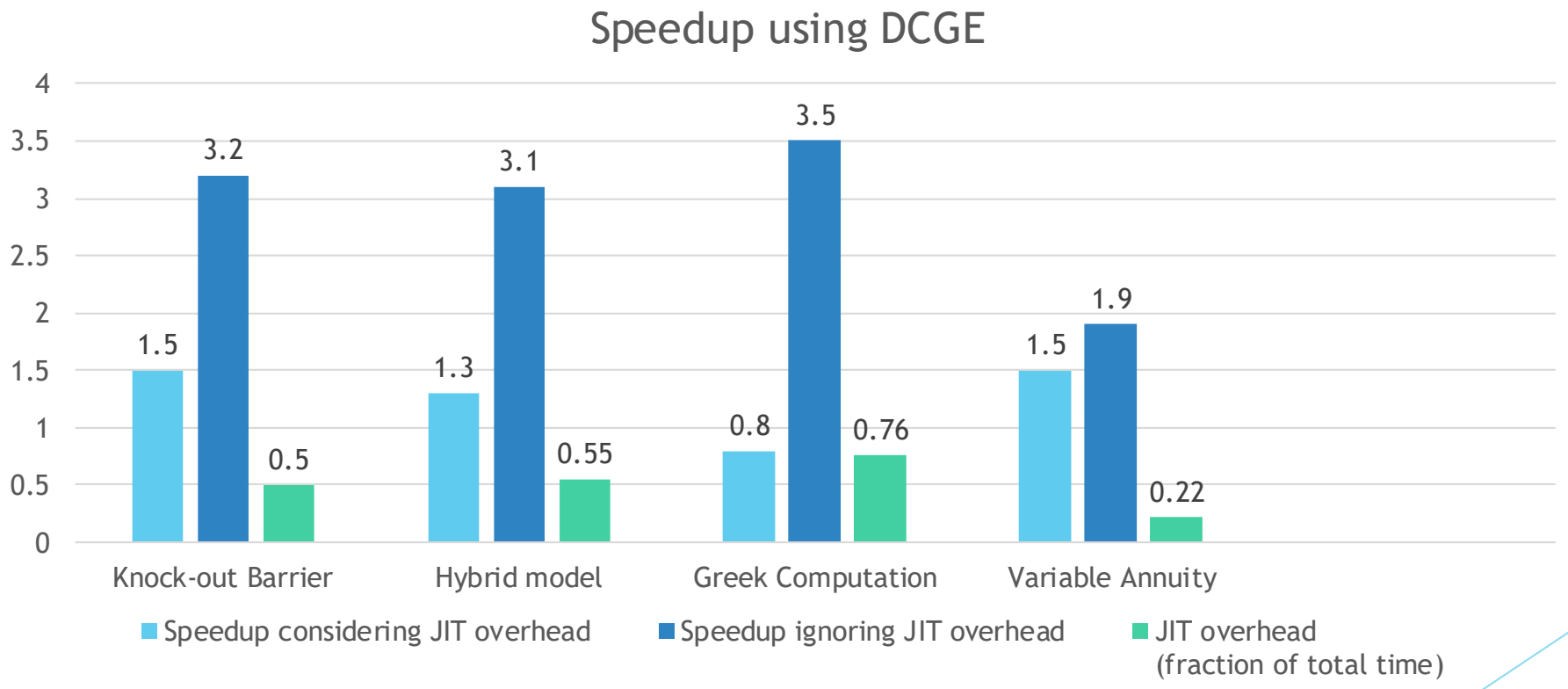
Benchmarks

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.

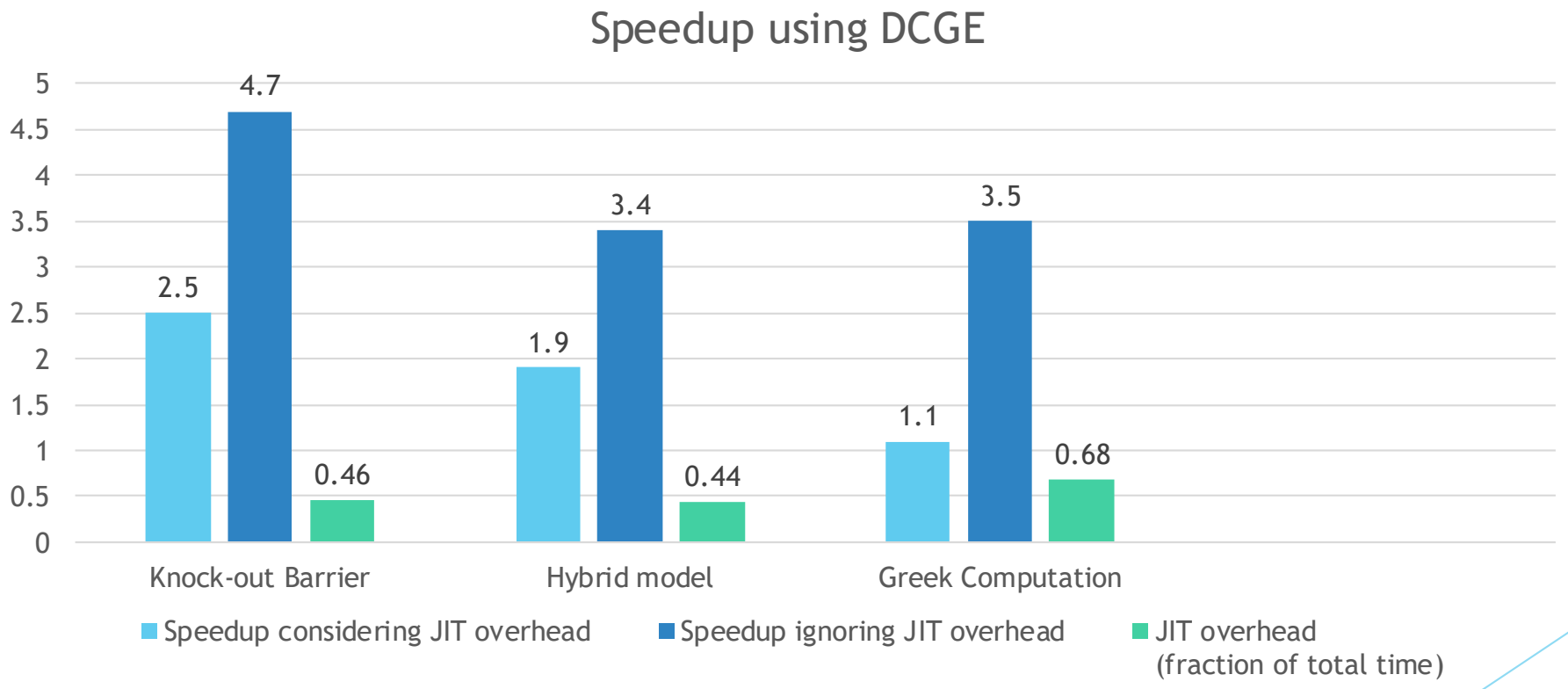
100k Monte-Carlo Paths



300k Monte-Carlo Paths



500k Monte-Carlo Paths



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

GPU TECHNOLOGY CONFERENCE

Questions?

Contact

vnagaraj@numerix.com
karmesin@numerix.com

Thank you

numerix

Backup Slide 1 – Execution times

Knockout Barrier

Number of Monte Carlo Paths	50000	100000	200000	300000	500000
No DCGE	0.096	0.160	0.324	0.469	0.801
DCGE	0.211	0.224	.239	0.294	.314
JIT overhead (part of DCGE)	0.151	0.162	0.155	0.150	.145

Hybrid Model

Number of Monte Carlo Paths	50000	100000	200000	300000	500000
No DCGE	5.94	9.9	20.0	29.3	49.5
DCGE	13.3	14.3	18.0	21.0	26.0
JIT overhead (part of DCGE)	10.6	10.4	11.1	11.7	11.6

Backup Slide 2 – Execution times

Greek Computation

Number of Monte Carlo Paths	50000	100000	200000	300000	500000
No DCGE	1.61	1.98	2.7	3.5	5.3
DCGE	3.4	3.6	3.8	4.2	4.7
JIT overhead (part of DCGE)	3.1	3.2	3.2	3.2	3.2

Variable Annuity

Number of Monte Carlo Paths	50000	100000	200000	300000	500000
No DCGE	45.2	85.5	162.7	244.6	-
DCGE	62.3	82.0	121.7	162.9	242.2
JIT overhead (part of DCGE)	37.1	37.0	37.3	37.3	37.5