ICHEC
Irish Centre for High-End Computing

Real-Time Risk Simulation

The GPU Revolution In Profit Margin Analysis

Gilles Civario gilles.civario@ichec.ie
Renato Miceli renato.miceli@ichec.ie

Science Foundation Ireland
Ireland's EU Structural Funds Programmes 2007 - 2013
Co-funded by the Irish Government and the European Union
EUROPEAN REGIONAL DEVELOPMENT FUND
An Roinn Oideachais agus Eolaíochta / Department of Education and Science
HEA
Higher Education Authority
An tÚdarás um Ard-Oideachas

Outline

- ICHEC in a nutshell
- The project context and customer requirements
- Technical environment and constraints
- Two case studies in porting
- Leveraging the lessons and community building
- Conclusion

ICHEC in a nutshell

- Irish Centre for High-End Computing
- The Irish HPC resources centre
- One keyword: Enablement
- GPU computing
  - Since 2009
  - 7th NVIDIA CRC
  - 2nd HMPP CoC

Technology transfer

- New activity
  - Started from a green field in October 2009
  - Many successes since then
  - Main activities
    - Consultancy
    - Training
    - Data mining and analytics
    - GPU acceleration

http://gpgpu.ichec.ie

The present project

- Disclaimer: details under strict NDA
  - No company name
  - No activity details
- But many interesting points still
  - London-based company
  - World leader in its sector
  - $5+ billion annual revenue
  - 35% sustained yearly growth over the past 10 years

Project constraints

- Optimise the computational part of a tool chain
  - Web based application for end-users
  - Written in Java and C#
  - Data updated in “real time”
  - Should give immediate result (500ms maximum computing time)
  - Should stay on one single server (no cluster allowed)
  - Monte Carlo type simulations of coming and/or on-going “events”
    - The more simulations run in the given time window, the better

→ Faster computation = Higher accuracy

Project goals

- Showcasing the potential of GPU acceleration
- Two simulators to port
  - The most simplistic one
  - The most complex one
- Integration to the production chain
- If successful
  - Training of the developers
  - Further in-house developments

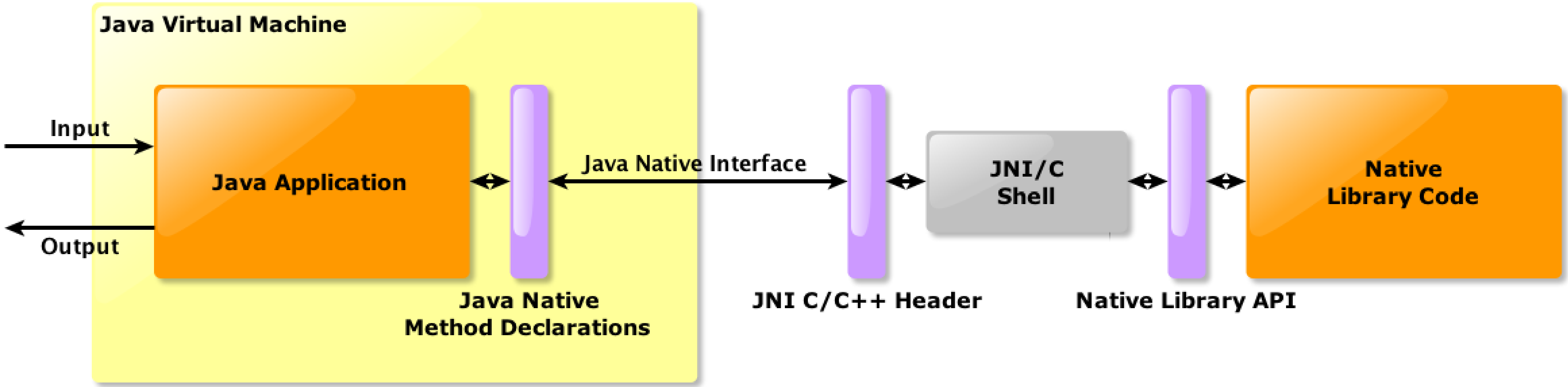
The first simulator

- The simplest simulator of the chain
  - 2000 lines of Java code (600 for computing)
  - Simulates events of fixed duration
  - Based on very wide tree random traversal
    - Weights based on collected statistics
  - Simulates all clock ticks but only collects statistics every 100 ticks
  - Results gathered as histograms
- Use: risk assessment for the occurrence of critical events

Porting strategy

- Keeping the Java front end
- Offloading the computational intensive part to native code with Java Native Interface
- Creating a C++ native version of it as a dynamic library
- Offloading the computationally intensive part to GPU with CUDA: one CUDA thread per simulation
- Collecting results from the GPU back to CPU and then to the Java virtual machine

Architecture



Stage 1: Native C++

- Data transfers of multi-dimensional Java arrays: data linearisation at the C++ level
- Ease of access to the linearised data: access macros (or C++ templates)
- CPU parallelisation of the code: OpenMP
- Random Number Generator: drand64
  - Thread-safe version drand64_r
  - Initialisation and possible bias?

Stage 2: CUDA code

- Maximisation of the caching potential
  - Constant data in `__constant__` memory
  - Lookup tables in `texture` memory
  - Histogram accumulations in `__shared__` memory
- Access macros to mimic multi-dimensional arrays
- The final code is almost identical to the initial Java one

  ➔The initial developers can maintain and evolve it easily

  ➔The code is not dead!

What it looks like

```
float getProb(int timeIndex1, int indexToUse, int timeIndex2){
    indexToUse = Math.max(Math.min(indexToUse, 80), 30);
    if (timeIndex1 >= Constants.getInstance().delimiter[timeIndex2][1] && timeIndex2 > 2){
        return Constants.getInstance().gProb
            [timeIndex1 - Constants.getInstance().delimiter[timeIndex2][1]]
            [indexToUse];
    } else
        return 0.0f;
}
```

JAVA

```
float getProb(int timeIndex1, int indexToUse, int timeIndex2){
    indexToUse = max(min(indexToUse, 80), 30);
    if (timeIndex1 >= my_Constants.get2D(delimiter, timeIndex2, 1) && timeIndex2 > 2){
        return my_Constants.get2D(gProb,
            timeIndex1 - my_Constants.get2D(delimiter, timeIndex2, 1),
            indexToUse);
    } else
        return 0.0f;
}
```

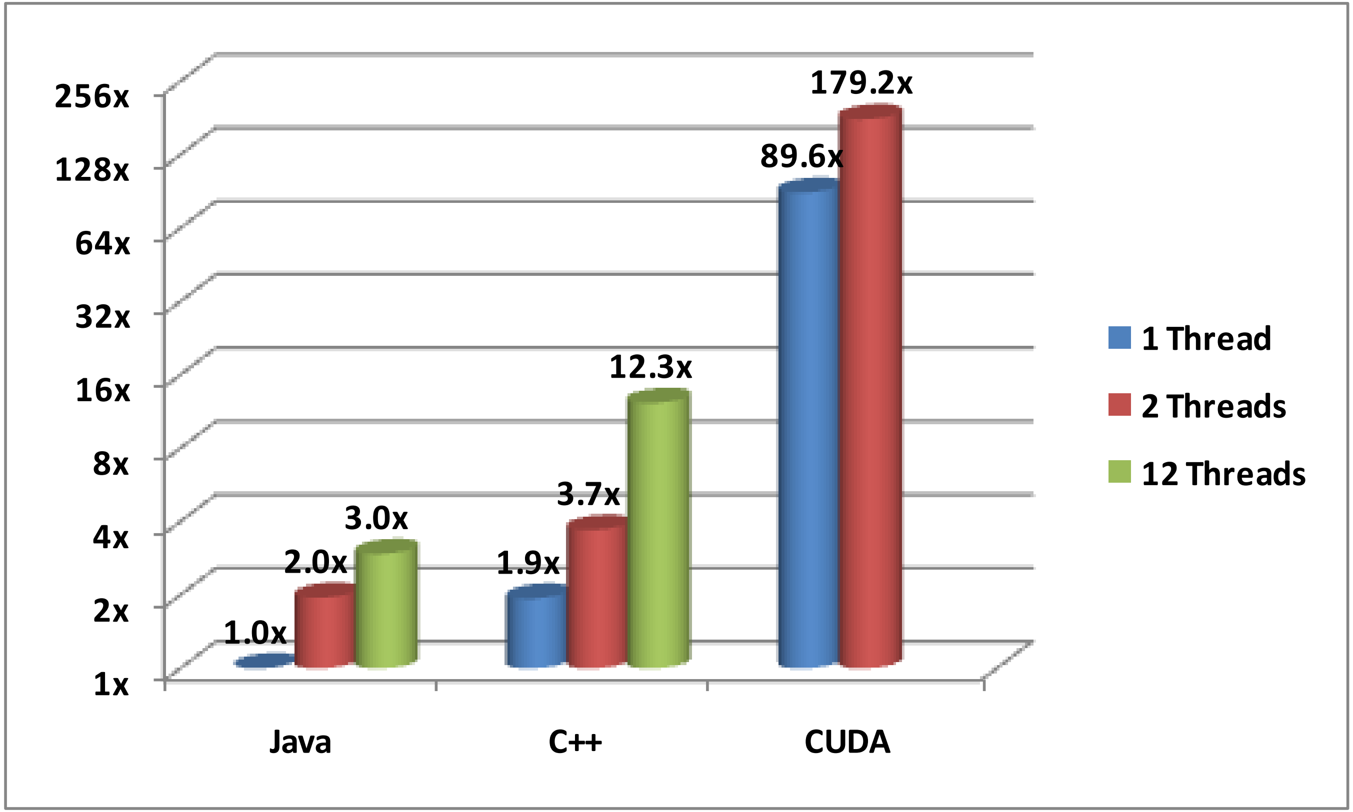
CUDA

Testing environment

- Development machine
  - 2 Intel Xeon X5650 Westmere @ 2.67GHz
    - 6 cores per CPU (state of the art at the time)
  - 2 NVIDIA Tesla C2050
    - 448 CUDA cores per GPU (state of the art at the time)
  - Linux Debian 6
    - Java 6.0 Sun JDK and OpenJDK
    - GCC 4.4 and 4.5 plus Intel C compiler 11.1
    - NVIDIA CUDA compiler 3.2 and 4.0

➔ Fair performance comparisons (no “cheats”)

Performance results



Main challenges

- Transferring data between Java and native code → JNI
- CPU level parallelisation → OpenMP
- Random number coherence → CURAND library
- Multi-dimensional lookup tables → texture memory

- Wide area to explore with one single thread per simulation → thread divergence?

The second simulator

- The most complex simulator of the chain
  - 4000 lines of Java (2500 for computing)
  - Freshly translated from C# and still buggy
  - Simulates events of fixed duration
  - Based on mathematical formulas
  - Simulates the whole event and collects a few statistics at the end
  - Results gathered as histograms
- Use: risk assessment for the occurrence of critical events

New challenges

- Still fresh → in depth refactoring and debugging
- Truly object-oriented programming → same approach in C++ and CUDA
- Computes in `double` rather than `float` → use of a versatile “real” type

- Intensive use of *log*, *exp*, *pow*, and *sqrt* → limited by registers?

Porting strategy

- Same as for the first code, but
  - Better integration within Java
  - Dynamic choice of back-end
    - Java: as initially
    - C++: native multi-CPU
    - CUDA: native multi-GPU
  - Impact assessment of precision for correctness and performance

➔90% of sources shared between C++ and CUDA

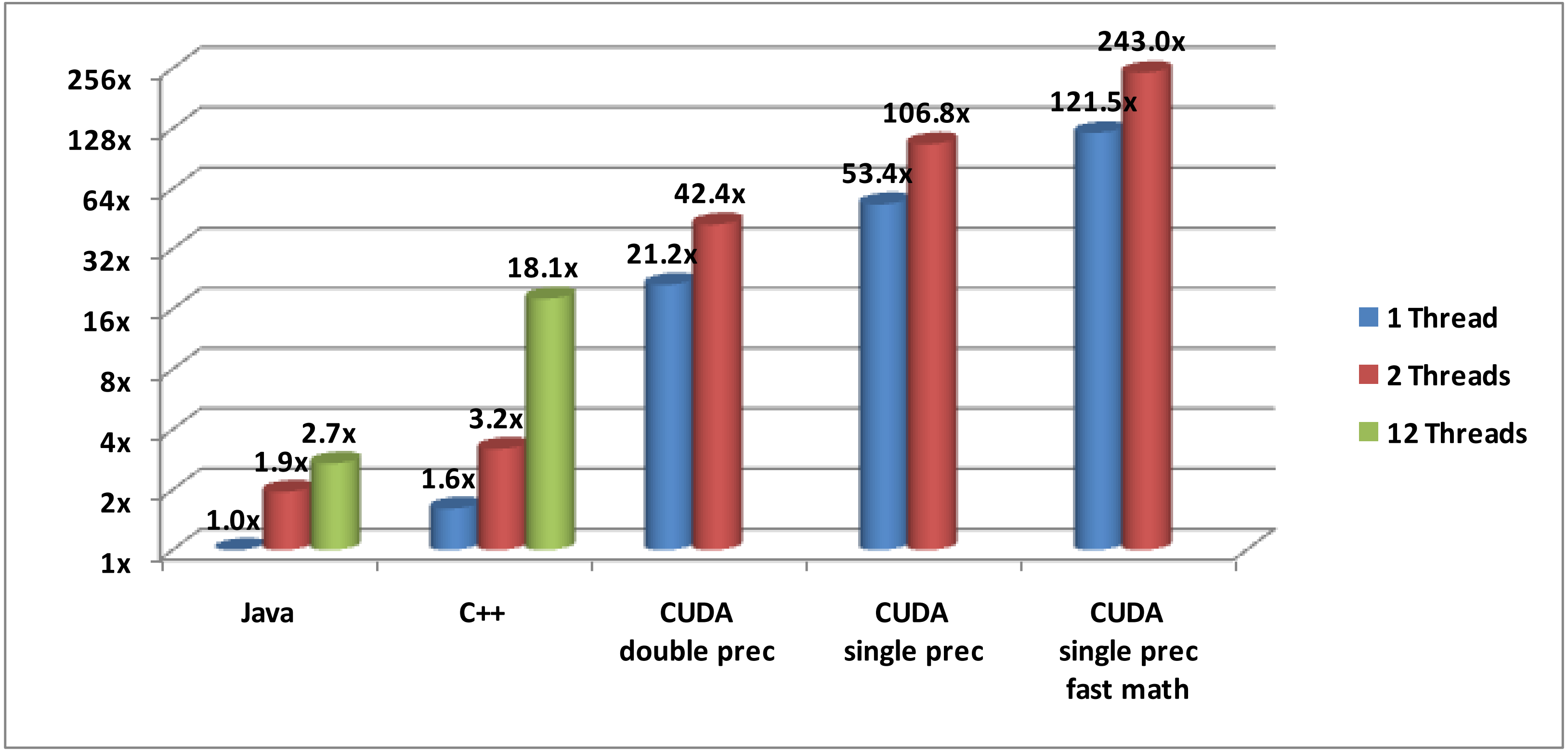
➔Final application of production quality

Remarks on optimisation

- Compute-bound code
  - Limiting factor: number of registers (spilled in local memory)
  - Increasing the L1 cache size gives a 40% boost in performance
- Need of **`atomicAdd`** for collecting the results
  - Not available for **`double`** (so far)
    - Software version: slow and sometimes deadlocking
    - No precision impact → use **`float`** for the corresponding data

What does it look like?

Performance results



Result analysis

- C++ version not much faster than the Java one
  - Compute bound with Java intrinsic functions already optimised
  - But much better parallelism than Java
- Precision impact
  - Double faster than float on CPU
  - Float faster than double on GPU
  - Almost no difference in precision for the results
  - Use of fast-math option very slightly changes the results in exchange of a 2.3x gain in performance

Current status

- Scalability tested with up to 8 M2090 cards
  - Per-card scalability C2050 → M2090 (c. +25%)
  - Codes scale almost linearly to the number of cards (7.6x for 8 cards)
- Tested with CUDA 4.1
  - Direct +10% for code 2
  - No change for code 1

➔ A whooping 840x for code 1 and 1100x for code 2

Follow-up: training

- Development of a CUDA training course
  - 3 days of training (lessons + labs)
  - NVIDIA-certified material
  - Certified CUDA programmers teaching
  - Possibility to deliver a certificate of completion at the end of the training
- 2 more days of pre-course training
  - Prerequisites for Linux and C++
  - Parallel algorithms and development

Leveraging
the Lessons Learned

Enhancements to Java

- JNI part
  - Mechanical: just do it
  - But error prone…
  - Could be automated
- Native part
  - Java translates in C++ almost directly
  - A few pitfalls, though…
  - Could be automated

Java2CUDA compiler

- OpenACC-like annotations for Java code
  - Compiles Java code straight to CUDA
    - Translates user-defined Java abstractions
    - Provides a GPU-aware Java API on the CUDA side
  - Accelerates to GPUs based on loop parallelization
- Aids developers to port code to GPUs
  - Provides code acceleration in no time
  - Avoids moving developers from their usual environment
  - Does not change programming paradigm (hides CUDA abstractions)

JThrust

- Framework for GPU programming in Java
  - Based on the NVIDIA Thrust library
  - Accelerates to GPUs based on routine calls
- Provides a C++ STL-like high-level programming API
  - General algorithms: sorting, randomising
  - Functional features: mapping, reducing, filtering
  - Data structures: lists, trees, maps
  - Why framework?
    JThrust can be extended with user-defined routines

JThrust: how should it work?

```
import ie.ichec.jthrust.*;

public class Snippet {
    public static void main(String[] args) {
        List<Integer> hostList = new HostList<Integer>();
        new RandomNumberGenerator<Integer>(
            RandomNumberGenerator.createRandomSeed()).generate(hostList);
        List<Integer> deviceList = new DeviceList<Integer>(hostList);
        new Sorter<Integer>(Sorter.DEFAULT_INTEGER_SORT).sort(deviceList);
        hostList.clean();
        hostList.addAll(deviceList);
        System.out.println(hostList);
    }
}
```

JThrust: current status

- A prototype is being developed
  - First working version should be released soon!
- Basic features come from the current Thrust lib
  - Want other features? Let us know!
- Have a Java application to be accelerated?
  - Let's test the "2X in 4 Weeks" thesis together ☺

Conclusion

- GPU computing for Java financial applications works!
  - Regardless of computational complexity
- Tremendous potential speed-ups
  - Same time window for 100x to 1000x more simulations
    ➔ Higher level of model precision
  - Allows to trade speed for model complexity
    ➔ Enable new models previously computationally unreachable
- The Java to CUDA translation process is sensitive
  - Java and CUDA are alike, but moving data around is critical
    ➔ New on-going developments to make it even simpler

Questions?

sfi Science Foundation Ireland

Ireland's EU Structural Funds Programmes 2007 - 2013
Co-funded by the Irish Government and the European Union

EUROPEAN REGIONAL DEVELOPMENT FUND

An Roinn Oideachais agus Eolaíochta
Department of Education and Science

HEA
Higher Education Authority
An tÚdarás um Ard-Oideachas