Incremental Risk Charge Calculation: A case study of performance optimization on many/multi core platforms

Amit Kalele*, Manoj Nambiar*# and Mahesh Barve*#
Center of Excellence for Optimization and Parallelization
Tata Consultancy Services Limited, Pune India

*kalele.amit@tcs.com, #m.nambiar@tcs.com, **mahesh.barve@tcs.com

Incremental Risk Charge calculation is a crucial part of credit risk estimation. This data intensive calculation requires huge compute resources. A large grid of workstations was deployed at a large European bank to carry out these computations. In this paper we show that with availability of many core coprocessors like GPU and MIC and parallel computing paradigms, speed up of order of magnitude can be achieved for the same workload with just a single server. This proof of concept demonstrates that with the help of performance analysis and tuning, coprocessors can be made to deliver high performance with low energy consumption, making them a “must-have” for financial institutions.

1. Introduction
Incremental Risk Charge (IRC) is a regulatory charge for default and migration risk for trading book position. Inclusion of IRC is made mandatory under the new Basel III reforms in banking regulations for minimum trading book capital. The calculation of IRC is a compute intensive task, especially the methods involving Monte-Carlo simulations. A large European bank approached us to analyze and resolve performance bottlenecks in IRC calculations. The timing reported on a grid of 50 workstations at their datacenter was approximately 45 min. Risk estimation and Monte Carlo techniques is well studied topic and details can be found in [1], [2], [3] and [4]. In this paper we focus on the performance optimization of the IRC calculations on modern day many/multi core platforms.

The modern day CPUs and GPUs (Graphic Processing Units) are extremely powerful machines. Equipped with many compute cores, they are capable of performing multiple tasks in parallel. Exploiting their parallel processing capabilities along with several other optimization techniques, could result in many fold improvement in performance. In this paper we present our approach for performance optimization of IRC calculations. We show that multifold gains, in terms of reduction in compute time, hardware footprint and energy required, can be achieved. We report that 13.5x and 5.2x speed up is achieved on Nvidia’s K40 GPUs and Intel KNC coprocessor respectively.

In this paper we present performance optimization of IRC calculations on Nvidia’s K40 [11] and Intel’s Xeon Phi or KNC coprocessors. We also present benchmarks on Intel’s latest available platforms namely Sandy Bridge and Ivy Bridge processors. The paper is organized as follows. In the next section (2), we briefly describe incremental risk charge in relations with credit risk. In section (3) & (4), we introduce a method for the IRC calculation along with the experimental setup and procedure. The performance optimization of IRC calculation on Nvidia’s K40 and
Intel’s KNC coprocessors are presented in section (6) & (7). We present our final experimental results and achievements in section (8).

2. Credit Risk & Incremental Risk Charge

Basel-III, a comprehensive set of reforms in banking prudential regulation, provides clear guidelines on strengthening the capital requirements through:

- Re-definition of capital ratios and the capital tiers.
- Inclusion of additional parameters into the Credit and Market Risk framework like IRC, CVA (Credit Valuation Adjustment) etc.
- Stress testing, wrong way risk and liquidity risk

The regulatory reforms and the ongoing change in the derivatives market landscape and the changing behavior of the clients are moving risk function from a traditional back office to a real time function. This redefinition of capital adequacy and requirements for efficient internal risk management has increased the amount of model calculation. This is required within the Credit and Market risk world and thus there is need for large scale computing. Incremental Risk Charge is one such problem we focused in this paper.

IRC calculation is crucial for any financial institutions in estimating credit risk. The IRC calculation involves various attributes like Loss Given Default (LGD), credit rating, ultimate issuer, product type etc. Standard algorithms as well as proprietary algorithms are used to calculate IRC and methods involving the Monte Carlo simulations are extremely compute intensive. In the next section we present one such algorithm and discuss computational bottlenecks.

3. Fast Fourier Transforms in IRC

IRC is a regulatory charge for default and migration risk for trading book position. One of the approaches based on Monte Carlo simulations for IRC calculation is described in below figure:

The data involved in default loss distribution is huge. In our case, the FFT computation for this data was offloaded to a grid of 50 workstations.

A typical IRC calculation for a single scenario involves computations of FFT for 160,000 arrays. Each array consisting of 32768 random numbers arising out of random credit movement paths. This translates to approximately 37GB of data to be processed for FFT computation. In all we have to process 133 such scenarios, which makes it a huge data and compute intensive problem. To summarize the overall complexity of the problem:

- 1 scenario of IRC calculation: 37GB of data
- Total scenarios: 133
- Total data to be processed: \((133 \times 37) = 4.9\) TB

To simulate the above computations, we carried out following procedure:

- For each IRC scenario
  - Create 160,000 arrays, each of 32768 elements
  - Each arrays is filled with random numbers between \((0 \sim 1)\)
  - Transfer the data in batches from host (server) to co-processors (Nvidia’s K40 GPU and Intel Xeon Phi or MIC) over PCIe bus
  - Compute FFT and copy back results

4. Experiment Environment

Following hardware setup and software libraries were used to carry out the above defined procedure. Performance analysis was done using Nvidia’s “nvvp” visual profiler tool.

The GPU benchmarks reported in this paper were carried out on the following system. This was enabled by Boston-Supermicro HPC labs, UK.

- Host: Intel Xeon E5 2670V2, 2 socket (10 core x 2), 64GB of RAM
- GPU: K40 x 4 (in x16 slot), 12GB RAM
- Freely available cuFFT library from Nvidia is used for FFT calculations [7], [8], [12].

Access to the Intel Xeon system was enabled by Intel India. All the experiments were carried out on following setup.

- Intel X5647 (Westmere), 4 core, 2.93 GHz, 24GB RAM
- Host: Intel Xeon E5 2670, 2 socket (8 core x2), 2.7 GHz processor, 64 GB of RAM
- Coprocessor: KNC 1.238 GHz speed, 16GB of RAM, 61 cores
- Intel MKL library is used for FFT calculations on host as well as coprocessor

In the following sections we discuss performance tuning of IRC calculations on various platforms.

5. IRC Calculations on Intel Westmere

We implemented the procedure explained in the previous section on Intel Westmere platform with Intel’s MKL math library.

The MKL is a collection of several libraries spanning linear algebra to FFT computations. The MKL provides various APIs for creating plans and performing different FFTs. Following APIs were used to perform 1D FFT in our exercise.

- \(\text{DftiCreateDescriptor(); DftiSetValue(); DftiCommitDescriptor();}\)
- \(\text{DftiComputeForward(); DftiFreeDescriptor();}\)
Since 4 cores were available for computation, a multi-process application was developed using Message Passing Interface (MPI) [5], [6]. The overall computations were equally divided among all the 4 cores. A code snippet of the main compute loop is given below in Figure 2.

```c
int num_arrays, nprocs, myrank;
int mystart, myend, range;
range = num_arrays/nprocs;
mystart = myrank * range;
myend = mystart + range;
for (i = mystart; i < myend; i++)
{
    load_data(buffer);
    DftiCreateDescriptor();
    DftiSetValue();
    DftiCommitDescriptor();
    DftiComputeForward(buffer);
    DftiFreeDescriptor();
}
(Figure-2)
```

Since FFT computation for each array is independent no communication was required among ranks over MPI.

It took 194 minutes to complete 133 IRC scenarios. It would require ~40 Westmere servers to complete these calculations in under 5 min mark. This adds too much cost in terms of hardware and power requirement. We hope to achieve a better performance with coprocessors and reduce hardware and power requirements. We consider Nvidia’s K40 and Intel’s KNC coprocessor in the following sections.

6. IRC on Nvidia K40 GPU

The K40 GPU is Nvidia’s latest Tesla series coprocessor. It has 2880 compute light weight GPU cores and rated at around 1TF of peak performance. Such platforms are extremely suitable for data parallel workloads. The cuFFT library was used for FFT computations. In this section, we describe the performance optimization in a step-by-step manner starting with a baseline implementation. Each step includes the measures taken in earlier steps.

- Baseline Implementation

Using the above mentioned procedure, a baseline implementation of FFT calculations was carried out. This involved creation of the appropriate arrays, calling the cuFFT functions for creating a 1D plan `cufftPlan1d` and `cufftExecR2C` for computing transform and finally copying the data back to host using `cudaMemcpy`.

It took ~67min to compute 133 scenario. We observed that the majority of the time (~61 min) is spent in data transfer between the host and the device. This data transfer happens over PCIe bus. Profiling the application using nvvp revealed that data transfer over PCIe bus was happening at only 2 – 3 GBps. The figure (3) below is a snapshot of the nvvp output.

The data is always transferred between pinned memory on host and device memory. Since normal allocation (using `malloc()`) is always a page-able memory, there is an extra step, which happens
internally, of allocating pinned memory and copying data between pinned memory and page-able memory.

- **Performance Optimization**
  The major performance issue observed was data transfer speed. We carried out couple of optimizations to resolve this issue. We discuss them below.

  - **Usage of Pinned Memory:** The data for 1 IRC scenario is approximately 37GBs. The data transfer rate achieved was poor since page-able memory was used. CUDA provides separate APIs to allocate pinned memory (`cudaHostAlloc` and `cudaMallocHost`). With pinned memory usage, we achieved a throughput of 5 – 6 GBps. A speed up of ~2.5x was achieved. The data transfer time of 133 IRC scenarios reduced to around 25 min and overall time was ~31 min.

  - **Multi Stream Computation:** In the current scheme of things, the data transfers and computations were happening sequentially in a single stream as shown in figure (5). By enabling multi stream computation, we could achieve two way overlap.
    - **Computations with data transfer:** GPUs have different engines for computations (i.e. launching kernels) and data transfer (i.e. cudaMemcpy). The computations were arranged in such a way that computations for one set and data transfer for next set happened simultaneously see figure (6).
    - **Data transfer overlap:** GPUs are capable of transferring data from host to device (H2D) and from device to host (D2H) simultaneously. With 4 streams, we could
achieve complete overlap between H2D and D2H transfer. With overlaps, further speedup of approximately 2.67x was achieved. The time for 133 IRC scenarios were reduced to ~11 min.

A single server can host multiple coprocessor cards. So within a box we could still enhance the performance by using multiple GPUs. This however had a limitation of data transfer bandwidth. Our experimental setup had 2 GPUs in x16 PCIe slot. The above optimized implementation was extended to use two GPUs. The final execution time obtained was 5.6 min. The below plot highlights step-by-step performance improvement.

![Step-by-step performance improvement on K40](image)

A marginal dip in the performance is observed which is attributed to the sharing of bandwidth for data transfer between host and multiple devices. The overall scale up achieved was close to 2x with 2 devices.

7. IRC on Intel KNC

Like NVidia’s K40 GPU, we also carried out the above exercise on Intel KNC coprocessor. The KNC was Intel’s first coprocessor with 61 cores and it also supports 512 bit registers for vector processing. These two feature together provides tremendous computing possibilities similar to NVidia GPUs. Intel also offers a highly optimized math library (MKL). The MKL is a collection of several libraries spanning linear algebra to FFT computations. However unlike cuFFT, MKL is not freely distributed. The MKL provides various APIs for creating plans and performing different FFTs. Following APIs were used to perform 1D FFT in our exercise.

- DftiCreateDescriptor(); DftiSetValue(); DftiCommitDescriptor();
- DftiComputeForward(); DftiFreeDescriptor();

Unlike GPUs, which only works in offload mode, the KNC coprocessor could be used for computation in native mode, symmetric mode and offload mode. In an offload mode, the main application runs on the host. Only compute intensive sections of the application are offloaded to the coprocessors. In native mode, full application run on the coprocessor and in symmetric mode both host and coprocessor run the part of application. In this exercise all the reading mentioned on KNC were taken in native mode. Only the final reading of the optimized code was taken in symmetric mode.
The biggest advantage of using KNC coprocessor, in the native mode, is no code level changes were required. The implementation done for Westmere platform was only recompiled for KNC platform. The overall computations were equally divided among all the 60 cores.

Each rank or core computed FFT for arrays in its range. This baseline code took 120 min for 133 IRC scenarios. Though the compute time was reduced as compared to Westmere platform (from 194 min to 120 min), the advantage is not as expected. We discuss the changes made to enhance the performance.

- **Performance Optimizations**

Since we were operating in native mode, no data transfer between host and coprocessor was involved. The MKL library used for FFT computation is highly efficient one. To identify performance issues, we referred to Intel’s guide to best practices [9], [10] on KNC and MKL. We exploited some of these techniques, which resulted in improved performance. We present these below:

- **Thread binding:** Multi-core coprocessors can achieve the best performance when threads do not migrate from core to core while execution. This can be achieved by setting an affinity mask to bind the threads to the coprocessor cores. We observed around 5 – 7 % improvement by setting the proper affinity. The affinity can be set by KMP_AFFINITY environment variable with the command:

  ```bash
  export KMP_AFFINITY=scatter,granularity=fine
  ```

- **Memory Alignment for input/output data:** To improve performance with data access, Intel recommends that the memory address for input and output data is aligned to 64 byte. This can be done by using MKL function `mkl_malloc()` to allocate input output memory buffer. This provided further boost of 7 – 9 % in the performance.

- **Re-using DFTI structures:** Intel recommends reuse of the MKL descriptor functions if FFT configuration remains constant. This reduces the overhead to initialize various DFTI structure. The MKL functions `DftiCreateDescriptor` and `DftiCommitDescriptor` allocates the necessary internal memory, and perform the initialization to facilitate the FFT computation. It may also involve the computation on exploring different factorizations of the input length and searching the highly efficient computation method. For the problem under consideration array sizes, type of data, type of FFT remains unchanged for the full application. Hence these descriptors can be initialized only once and then reused for all the data. Initializing these descriptors only once outside the main compute loop gave the desired ~3.6x performance gain.

With all the above changes in place, we observed significant improvement in the performance of IRC calculations. Timing for 133 IRC scenario was reduced to approximately 32 min from 120 min.

Similar to GPUs, a single server can host multiple KNC coprocessor. Since such a setup was not available, we expect that it would take around 16 – 17 min for IRC calculations on 2 KNCs.
8. Final Results

In the earlier sections we discussed performance optimization of IRC calculations on Nvidia’s K40 and Intel’s KNC coprocessor. Both the platforms are capable compute resources having their own pros and cons. In this section we summarize overall achievements and other benefits enabled by this optimization exercise.

- **Execution time with Hybrid computing**
  Several fold performance improvement was achieved on both coprocessors. All the workload was taken by coprocessors. However the host machine could be utilized to share the partial workload. In case of Intel KNC, it required only recompilation of the code to facilitate this. However for K40, we had to rework the code to accommodate these changes. This was achieved by combining MPI and CUDA C.

  - 35 – 40 % further speed up was achieved on both KNC and K40 by enabling the workload sharing.
  - Out of 160,000 arrays per IRC scenarios, 60,000 were processed on each of K40s and 40,000 on host. In case of KNC the split up was 30,000 on each of KNC and 100,000 arrays on the host.

The figure (8) summarizes the best results achieved with hybrid computing on coprocessors along with other Intel platforms.

![Figure 8 IRC Performance comparison across all platforms](image)

Clearly K40 GPUs performs better than KNC coprocessor. However KNC offers ease of programming. Any x86 applications will only require a recompilation to work on KNC. On the other hand porting application to K40 requires lot of programming efforts in terms of CUDA C.

- **Energy Consumption**
  The energy required to carry out the computations directly affects the cost of the computation. In our experiment, the K40 performed best. Assuming this as benchmark, we rationalize the hardware requirement of other platforms to achieve the same performance and intern calculate
the energy required to carry out the computation. The energy consumptions are computed considering the rated wattage for each Intel and Nvidia platform.

The figure (9) show that drastic reduction in compute time and cost for computation is achieved by optimizing IRC calculations on both the platforms. But the gains are not only limited to these factors. This exercise also enabled huge reduction in hardware footprint, data center floor space and ease to maintain compact system.

![Figure 9 Energy requirement for best performance for all platforms](image)

9. Conclusion

This paper highlights the importance of optimizing an application for a given platform. The baseline results suggests that simply using the new hardware with libraries would almost always results in a suboptimal performance. The modern day many core GPUs or Coprocessors have tremendous computing capabilities. However new and legacy applications could achieve huge gains only by doing optimization by detailed analysis and measurement with proper profiling tools. In this paper we highlighted the above fact with an example of IRC calculations. Though the chosen application was from financial risk computation, compute intensive applications from various domains can benefit by performance optimization with many core parallel computing.

The highlights of the work are as follows:

- With the optimizations, we achieved approximately 13.5x and 5.2x speedup on K40 and KNC respectively for IRC calculations and ~150x reduction in energy consumption.
- Hybrid computing utilizes both, the host and the coprocessor and intern gives best performance.
• These high performance setups (coprocessor HW + optimized applications) would allow banks or financial institution to simulate many more risk scenarios in real time and enable better investment decisions.

We conclude this paper on a note that, with proper performance optimizations, the many/multi core parallel computing with coprocessors enable multi-dimensional gains in terms of reduction in compute time, cost of computations and hardware foot print.

Acknowledgement

The authors would like to thank Jack Watts from Boston Limited (www.boston.co.uk) and Vineet Tyagi from Supermicro (www.supermicro.com) for enabling access to their HPC labs for K40 benchmarks. We are also thankful to Mukesh Gangadhar from Intel India for enabling access to Intel KNC coprocessor.

References


