MATLOGICA

An elegant approach to run existing CUDA analytics on both GPU and CPU, with added benefit of AAD

Executive Summary

GPUs have long been seen as a silver bullet for financial organisations striving to achieve top computational performance. In pursuit of the proclaimed 100-1000x GPU performance gains against CPU, many have taken a costly choice to transition their analytics to CUDA.

In some cases this decision was made almost a decade ago. Since then CPU based systems have made a leap in parallel compute capacity and are now comparable, sometimes exceeding GPU systems when total cost of ownership is accounted for.

In this post we present a solution that allows organisations with existing CUDA projects to assess performance loss(or gain) for transitioning from GPU to modern CPU systems. Using real-life CUDA examples, we demonstrate how existing GPU-only code can be adopted to run on CPU or GPU at the same time. This should allow companies to make fair assessments of performance provided by both technologies.

Developing analytics for CPU usually requires less effort and allows for advanced techniques such as Automatic Adjoint Differentiation (AAD). Companies have to make a difficult decision accounting for all pros and cons of both technologies.

We also make available an open source equity pricing model benchmark implemented for both CPU and GPU to facilitate practitioners to help extract top performance from both platforms and estimate unbiased metrics.

Introduction

When GPU initially came to the market, Mike Giles, a renowned quantitative finance influencer and major promoter of GPU for financial computations commented: "If there is a big enough market, someone will develop the product". After a decade or so of technological development, the anticipated revolution for the quantitative finance world has not happened and does not seem to be getting closer. When Mike Giles was asked "**Will GPUs have an impact in finance?**" he replied, "I think IT groups are keen, **but quants are concerned about the effort involved… quants have enough to do without having to think about CUDA programming"** [MG, pp.22,37]. Moreover, using GPU comes with technical limitations, such as strict memory volume constraints.

For many years, the crucial factor in favour of GPU was the ability to generate kernels that can be safely processed in parallel. With proclaimed performance gains of 1000x, a CFO might be persuaded to switch to GPU, despite the significant investment required to transition to CUDA and subsequent higher software support costs.

However, according to the most trustworthy and impartial benchmark (STAC-A2), when hardware manufacturers put maximum software development effort to extract top performance from their offerings, CPU and GPU go neck-and-neck.

In the example below we provide an approximate comparison of performance and operational costs of modern CPUs vs GPUs, using cloud costs as a proxy for owning such a set up. It shows that the average cost of a CPU TFLOP is ~<30% higher than GPU. Therefore, the maximum theoretical saving for a CFO is about 30%, not 1000x!

 info@matlogica.com

Feature	NVIDIA GPU V100	Intel® Xeon® Platinum 9282 Processor
Number of cores	5,120	56 x 2
Clock frequency	877 MHz	2.6 GHz
Operations per clock (float precision)	1	32
FMA	2	2
TFLOPS (all of above multiplied)	8.98	18.64
Approx monthly cost (GCP)	1300$	3416$
Approx monthly cost, per TFLOP	145$	183$

Other key considerations include the software redesign effort, and the increased support and maintenance costs associated with CUDA, which are driven by the specialised nature of the code and thus requiring specialised developers. In addition, GPU vendor-lock is likely to drive an incremental cost increase as older generations of hardware become outdated.

The performance gained from transitioning from CPU to GPU can't be fully explained by the change of hardware. It also involves a costly and easy-to-dismiss change of mindset, due to the transition from object-oriented languages to matrix-vector multiplication paradigm, that would yield performance improvements despite chip architecture.

Many large banks made a long-term commitment to CUDA/GPU a few years ago. Some have come to realise that this decision has in fact created a raft of new liabilities; including, high maintenance costs, scarcity and thus difficulty to find and recruit new talent, hardware reaching end-of-service and so becoming obsolete, as well as hitting the technical limits of GPU due to new business needs. However, there is a way out of the vendor-lock imposed by this migration to CUDA/NVIDIA.

Until now, , technology similar to CUDA, allowing and supporting safe multithreading was unavailable on a CPU. In response, MatLogica has developed AADC. Unlike CUDA, AADC can use existing C++ object-oriented code to generate optimised kernels for scalable execution on a CPU with minimal effort from developers.

AADC is able to simply **reuse existing CUDA analytics, implemented for GPU,** and run it on scalable CPUs instead. With minimal changes, existing CUDA code can be adapted for AADC and executed using multi-threading and vectorization on a CPU to get top performance. Unlike GPU, CPU has plenty of memory to solve large problems and support AAD!

Idea: Using AADC to generate scalable CPU kernels with EXISTING CUDA analytics

CUDA mainly uses C++ syntax and adds some extensions relevant to parallel programming and GPU management. The AADC approach is to record scalable CPU kernels by executing original

user code for one data sample (for instance, one MonteCarlo path). By getting CUDA analytics to run with AADC for one data sample on CPU, we can record the full valuation graph and therefore compile scalable CPU kernels that support execution in a safe multithreaded environment, whilst also taking advantage of AVX native CPU vector arithmetics.

More complex problems, such as American Monte Carlo pricing and xVA, can be handled with a similar approach albeit with modest increases in the complexity of the code.

How: Going back to host

To run the existing CUDA code with AADC on the CPU we disable CUDA extensions to make the code compatible with the standard C++ compiler and ready for AADC kernel compilation. For demonstration purposes we are very explicit here. In real life projects this code can be wrapped for simplified use.

New compilation unit for AADC on CPU may look like this :

```
#define double idouble                  change native types to active AADC types
#define bool ibool                      to take advantage of operator overloading

// Override CUDA extensions:

#define __global__                      ignore __global__
void __syncthreads() {};                provide simple stub implementation for CUDA specific
                                        API. Other methods can also be implemented such as
                                        CUDAMemGetInfo etc.
struct { int x = 0; } threadIdx;        use the zero-th thread to record MC path 0
struct { int x = 0; } blockIdx;
struct { int x = 0; } blockDim;

#include "kernel.cu"                    Original user CUDA kernel

// Revert back overrides:

#undef double
#undef bool
#undef __global__

// Normal C++ code follows here
```

After applying these fixes, kernel.cu should compile as normal C++.

We can now add the AADC kernel compilation and the execution driver as with any other C++ code. This normally consists of 2 steps:

1. Starting kernel compilation and execution analytics from kernel.cu. For this we need to explicitly identify model inputs and outputs;
2. Use the compiled CPU kernel instead of the original function for subsequent Monte-Carlo iterations and running simulation across multiple CPU cores and avx2/avx512 parallelization.

Example: Equity Derivative Pricing

We use the model for pricing a single-asset Equity Linked Security option developed for CUDA/GPU to examine the changes required to enable execution on a CPU, using MatLogica's AADC.

The original code is taken from https://github.com/ymh1989/CUDA_MC and is inspired by https://www.quantstart.com/articles/Monte-Carlo-Simulations-In-CUDA-Barrier-Option-Pricing/

The source code can be built on Linux and Windows and is available in "CUDA_Example/AADC_Enabled/one-asset ELS/code" and the user manual is available as "Manual.pdf". For the vanilla option pricer, no changes to "kernel.cu" are required. For path-dependent ELS option, minimal changes were needed and GPU/CPU compatibility is maintained.

The source code can be obtained on request using MatLogica website.

What about performance?

Let's compare the performance of pricing a one-asset Equity Linked Security option using CUDA/GPU and the AADC-enabled version of CUDA code, on CPU. This is a path-dependent option that requires 1080 timesteps and performs 100k Monte-Carlo simulations. Performance measures only include process simulation and pricing logic (random number generation is excluded).

Machine	Price (using google cloud as a proxy)	Execution time*
NVIDIA V100	1300 USD	10.2 ms
CPU, 30 threads + avx512	915 USD	13.5 ms

*The results are preliminary and are being validated by the hardware vendors.

Based on these results, we get comparable performance between top-of-the-line GPU/CUDA and AADC-adapted CUDA code on a CPU. The changes required for CUDA code are minimal. Apart from integrating MatLogica AADC, no additional optimisations were performed.

This example is open source and anyone can run it themselves as well as recommend improvements for both CPU and GPU. We will update this tablet as we receive feedback from hardware manufacturers and developers.

Conclusion

CUDA is not a one-way street. With minimal changes, it is possible to run CUDA code on a scalable 64bit CPU and take advantage of AAD as an additional benefit. We have shown that it is reasonably simple to support existing CUDA projects for dual CPU and GPU builds. This allows organisations to make informed decisions about hardware options and choose the best option depending on business needs.

In this post, we used an example of an embarrassingly parallel pricing method. In MatLogica, we have solutions for a wide range of more complex models typical to quantitative finance such as Longstaff-Schwarz pricing of callable products, XVA, PDEs, etc.