THE MULTI-LEVEL SPARSE GRID INTERPOLATION KERNEL WITH COLLOCATION (MUSIK-C) ALGORITHM, APPLIED TO BASKET OPTIONS USING C++ AND CUDA

NICHOLAS WILTON

Contents

1. Abstract

The recently developed MuSiK-c algorithm with promising characteristics for solving higher-dimensional Partial Differential Equations was converted into C++ and the GPU language CUDA. This new implementation was successfully verified against the original MatLab prototype solving Black-Scholes for the European Call Option. MuSiK-c was further extended to handle Multi-dimensional Basket Options, however successful verification was only achieved for a single underlying Asset. Without significant micro-optimisations the performance of the algorithm was comparable to the original when running solely on the CPU. For the case of the GPU promising results were obtained but significant further work needs be done before it matches and out performs a CPU only deployment.

2. Introduction

Numerical Analysis is the study and use of computation algorithms in approximation within the field of mathematical analysis of which (and in particular interest to this paper), Partial Differential Equations (PDEs) are common throughout many areas of science. Whilst analytic solutions exist for some PDEs using such techniques as 'seperation of variables', 'integral transform' and 'change of variables', often either no solution exists orthe analytic tools are currently insufficient to find it.

As well as Stochastic methods such as Montecarlo, the known algorithms developed for solving PDEs are generally classified into nine main methods:

1 Finite Difference.
2 Method of Lines.
3 Finite Element.
4 Gradient Discretisation.
5 Finite Volume.
6 Spectral.
7 Mesh-free.
8 Domain Decomposition.
9 Multi-grid.

Some recent work in the field draws from ideas contained within a number of these methods to address the *curse of dimensionality* in high-dimensional PDEs. A collection of papers from the University of Leicester[1] [2] [3] derives techniques based around the concept of Sparse Grids (7,8,9), Collocation, Multi-Level Collocation (9) and Radial Basis Functions (6).

The aim of this paper is to build upon this work, take some of the proposed algorithms that have been demonstrated in MatLab in 1-dimension; and using a more versatile and scalable technology expand them to the N-dimensional case; whilst demonstrating how recent improvements in technology can aid in paralellisation of such algorithms. The main reference of this work is Y. Zhao's PhD Thesis (2017) [1] and much of the accompanying theory and source code presented here was either derived or inherited from that particular paper.

Before discussing the technical aspects of this implementation, a little theoretical background is needed.

3. Theoretical Background

Whilst an in-depth discussion of the theoretical background to this work can be found in [1], here we present a more brief overview of the main points.

3.1. **Partial Differential Equations.** Two particular types of PDEs that occur often in nature and are not always suitable in higher dimensions to be solved with analytic methods are Elliptic and Parabolic PDEs. Here we are mostly concerned with how such equations can be treated using a particular Spectral method involving Radial Basis Functions.

3.2. **Elliptical PDEs.** If we define a linear operator $L : C^2(\Omega) \rightarrow C(\Omega)$ as an elliptic differrential operator on u(x) as:

$$Lu(x) = \sum_{i,j=1} a_{i,j}(x)\frac{\partial^2}{\partial x_i \partial x_j}u(x) + \sum_{i=1} b_i(x)\frac{\partial}{\partial x_i}u(x) + b_0(x)u(x) \tag{1}$$

where the coefficient matrix $\left[a_{ij}(x)\right] \in \mathrm{Re}^{dxd}$ satisfies the condition:

$$\exists\ \alpha > 0$$

such that,

$$\sum_{i,j=1}^{d} a_{ij}(x)c_i c_j \geq \alpha ||c||_2^2$$

for all $x \in \Omega$ and $c \in \mathrm{Re}^d$

Then for a boundary value problem (i.e. where the problem domain is bounded by some function such as a skipping rope attached to a wall), we would then solve the second order elliptic PDE with the boundary conditions:

$$Lu = f \text{ in } \Omega \tag{2}$$

$$u = g \text{ on } \partial\Omega \tag{3}$$

Where again, L is the elliptic operator and f and g are the functions describing the boundary.

3.3. **Parabolic PDEs.** We can represent parabolic PDEs (for which the Black-Scholes equation is a 2nd order example), in a similar manner. Where L is a linear operator on the domain of Ω such that:

$$Lu(x,t) = u_t \sum_{i,j=1} a_{i,j}(x,t)\frac{\partial^2}{\partial x_i \partial x_j}u(x,t) - \sum_{i=1} b_i(x,t)\frac{\partial}{\partial x_i}u(x,t) - c(x,t)u(x) \tag{4}$$

Which for Black-Scholes in N-dimensions is:

(5)

$$LV(S,t) = \frac{\partial V}{\partial t} + \frac{1}{2}\sum_{i=1}^{d}\sum_{j=1}^{d}\rho_{ij}\sigma_i\sigma_j S_i S_j \frac{\partial^2 V(S,t)}{\partial S_i \partial S_j} + \sum_{i=1}^{d}(r - q_i)S_1\frac{\partial V(S,t)}{\partial S_i} - rV(S,t)$$

To solve parabolic equations with Radial Basis Functions, Zhao uses two methods, The Method of Lines (MoL) and the Space-Time method of Myers incorporating Kansa's approach to elliptical PDEs. A brief overview of both will be outlined in the next sections whilst an in-depth discussion can be found in Zhao's original thesis [1]

3.4. **Radial Basis Functions.** A radial basis function (RBF) is a real valued function whose value only depends on it's distance from the origin or centre such that:

$$\phi(x, c) = \phi(||x - c||) \tag{6}$$

Where x is the point of interest and x_i is the location of the central point. Some examples of RBFs are the Euclidean distance:

$$\phi(r) = \sqrt{x^2 + y^2} \tag{7}$$

Hardy's Multiquadric RBF:

$$\phi(r) = \sqrt{c^2 + ||x - x_i||^2} \tag{8}$$

The Gaussian RBF

$$\phi(r) = e^{-||x-x_i||^2/c^2} \tag{9}$$

Now, if we want to approximate some function $u(x)$ for a set of scattered data points say $X = x_1, x_2, ...x_N \subset \mathrm{Re}^d$ we would use interpolation via:

$$\hat{u}(x) = \sum_{i=1}^{N} \lambda_i \Phi(||x - x_i) \tag{10}$$

before then solving the linear system:

$$A\lambda = y \tag{11}$$

where,
$A_{j,k} = \phi(x_j)$, for j,k=1,2,...N
$\lambda = [\lambda_1, ...\lambda N]^T$
$y = [y_1, ...yN]^T$

As the inverse of A exists, then we can be sure that a unique solution exists.

Of particular interest are the Multiquadric (MQ) and Gaussian RBFs as they are both infinitely differentiable and have been shown to exhibit accuracy, stability and ease of implementation in for instance Franke (1982)[7]. As such they have become popular in the literature of various interpolation schemes of which an example is A. Pena's 8-step scheme (2005) [10]. In both cases a parameter c is defined, known as the *shape parameter* the size of which will sharpen (decreasing c) or flatten (increasing c) the function (see figure 1).

It has been shown in T. A. Driscoll and B. Fornberg.(2002) [8] (amongst others), that a larger value of c will increase accuracy but exceed a limit and the system will become ill conditioned and unstable. Likewise reduceing c will improve the conditioning but also lead to an inaccurate solution. There has been significant effort devoted to finding the optimal value of c for different RBFs however this is still considered an solved problem within the field [1].

3.5. **Ansiotropic Radial Basis Functions.** If the domain of interest is not the same size in all dimensions then an RBF becomes ansiotropic. To model this let $\phi(||\dot{-}x_i)$ be some RBF centred around $x_i \in \mathbb{R}^d$ and $A \in \mathbb{R}^{dxd}$ is an invertible transformation matrix, then the ansiotropic radial basis function ϕ_A is defined by:

$$\phi_A(||\cdot - x_i||) = \phi(||A(\cdot - x_i||) \tag{12}$$

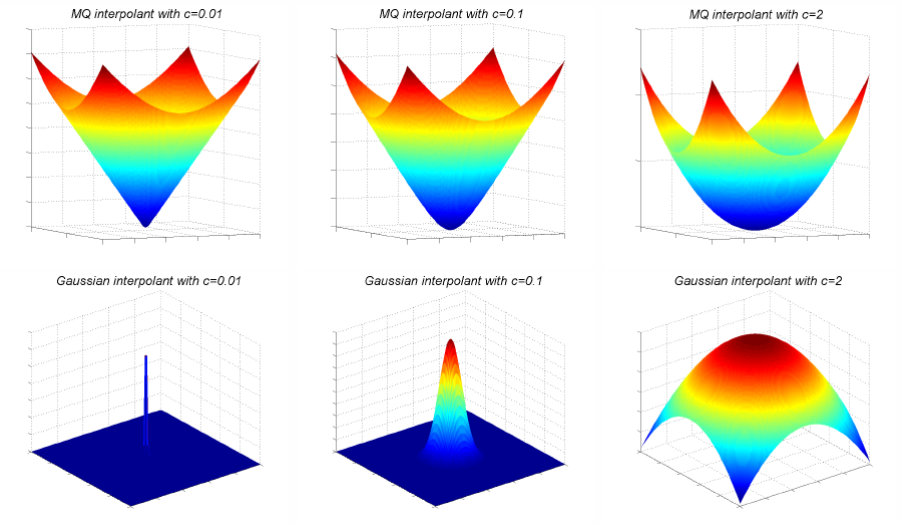

FIGURE 1. The Effect of Shape Parameter c on the Gaussian and Multiquadric Radial Basis Functions
Source: Y. Zhao (2017) [1] page 21

Furthermore we can then define the Ansiotropic Tensor Based Product Function (ATBPF) of the MQ and Gasussian basis functions respectively, as:

$$\phi_{A,x_i}(x) = \prod_{k=1}^{d} \sqrt{A_k^2(x_k - x_i^k)^2 + c_k^2} \tag{13}$$

$$\phi_{A,x_i}(x) = \prod_{k=1}^{d} \exp\left\{\frac{-A_k^2(x_k - x_i^k)^2}{c_k^2}\right\} \tag{14}$$

Where, k is the k^{th} dimension of x and A_k is k^{th} diagonal element of $A \in \mathbb{R}^d x \mathbb{R}^d$.

Whilst we can observe that the Gaussian ATBPF still belongs to the family of RBFs, the MQ ATPBF is no longer radially symmetric.

Now, if we let $Ch_k = c_k/A_k$ we find...

$$\phi_{A,x_i}(x) = \prod_{k=1}^{d} \sqrt{(x_k - x_i^k)^2 + Ch_k^2} \tag{15}$$

$$\phi_{A,x_i}(x) = \prod_{k=1}^{d} \exp\left\{\frac{(x_k - x_i^k)^2}{Ch_k^2}\right\} \tag{16}$$

... that Ch_k represents a 'shape parameter' as noted previously where h_k represents the distance between nodes in the kth direction and A_k is the number of nodes in that same direction minus one.

The first and 2nd derivatives of the ATPBFs are then in the Mutliquadric case,

$$D_{x_p}(\phi_{A,x_i}) = \frac{x_p - x_i^p}{\sqrt{(x_p - x_i^p)^2 + (Ch_p)^2}} \prod_{k\neq p}^{d} \sqrt{(x_k - x_i^k)^2 + Ch_k^2} \tag{17}$$

$$D^2_{x_p}\phi_{A,x_i}(x) = \frac{(Ch_p)^2}{[(x_p - x_i^p)^2 + (Ch_p)^2]^{3/2}} \prod_{k\neq p}^{d} \sqrt{(x_k - x_i^k)^2 + Ch_k^2} \tag{18}$$

and in the Gaussian case

$$D_{x_p}(\phi_{A,x_i}) = -\frac{2(x_p - x_i^p)}{(Ch_p)^2} \prod_{k=1}^{d} \exp\left\{ \frac{(x_k - x_i^k)^2}{Ch_k^2} \right\} \tag{19}$$

$$D^2_{x_p}\phi_{A,x_i}(x) = -(\frac{2}{(Ch_p)^2} + \frac{4(x_p - x_i^p)^2}{(Ch_p)^4}) \prod_{k=1}^{d} \exp\left\{ \frac{(x_k - x_i^k)^2}{Ch_k^2} \right\} \tag{20}$$

It can be shown (Y. Zhao (2017) [1] p25-26), that for higher dimensional problems we can use the tensor product of ATBPFs to find:

$$(\Phi)^{-1} = (\Phi_1)^{-1} \otimes ... \otimes (\Phi_d)^{-1} \tag{21}$$

$$\lambda = (\Phi)^{-1} \cdot y \tag{22}$$

Here, Zhao makes some important points: Inverting Φ matrix would cost $O(N^{6d})$ For Φi this becomes $O(N^6)$ and the Kroencker Product is $O(N^{2d})$

Therefore potentially using ATPBFs produces some good performance benefits over and above summing RBFs when scaling in multiple dimensions.

Whilst Zhao also discussed convergence he also notes there is no current theoretical proof of the convergence of MuSiK-c however his numerical experiments do demonstrate that it does.

3.6. **Kansa method.** Kansa's method (E. J. Kansa. (1990)[5][6]) is a spectral method of approximating u(x) via:

$$\hat{u}(x) = \sum_{i=1}^{N} \lambda_i \Phi(||x - x_i) \tag{23}$$

Where Φ is the radial basis function of choice.

For Kansa's method we choose a $\Xi = \Xi_1 \cup \Xi_2$ which we will call central nodes and where $\Xi_1 \in \Omega$ (i.e are interior points) whilst $\Xi_2 \in \partial\Omega$ exist on the boundary denoted by $\partial\Omega$.

The key of course, is to find the λ coefficients for each of the summation terms that make the best approximation of the function u(x). For the elliptical PDE of (1) we can substitute (23) into the boundary counditions (2) and (3) to get:

$$\sum_{i=1}^{N} \lambda_i L\phi_{x_i}(x_j) = f(x_j), \text{for j = 1,2,...n} \tag{24}$$

$$\sum_{i=1}^{N} \lambda_i \phi_{x_i}(x_j) = g(x_j), \text{for j = n+1,n+2,...N} \tag{25}$$

Which as a matrix system can be represented as:

$$\begin{bmatrix} L\phi_{A,x_1}(x_1) & L\phi_{A,x_2}(x_1) & ... & L\phi_{A,x_N}(x_1) \\ L\phi_{A,x_1})(x_2) & L\phi_{A,x_2}(x_2) & ... & L\phi_{A,x_N}(x_2) \\ \vdots & \vdots & \vdots & \vdots \\ L\phi_{A,x_1})(x_n) & L\phi_{A,x_2}(x_n) & ... & L\phi_{A,x_N}(x_n) \\ \phi_{A,x_1})(x_{n+1}) & \phi_{A,x_2}(x_{n+1}) & ... & \phi_{A,x_N}(x_{n+1}) \\ \vdots & \vdots & \vdots & \vdots \\ \phi_{A,x_1})(x_N) & \phi_{A,x_2}(x_N) & ... & \phi_{A,x_N}(x_N) \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_n \\ \lambda_{n+1} \\ \vdots \\ \lambda_N \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \\ g_{n+1} \\ \vdots \\ g_N \end{bmatrix}$$

Whilst much work using collocation with RBFs had been performed using Kansa's method to solve elliptic boundary value problems, it wasn't until Myers et al. [4] proposed the space time method that applications for parabolic problems were first successfully investigated.

3.7. **Myer's Space-time method.** In a parabolic problem with a spacial domain $\Omega \in \mathbb{R}^d$ and a time domain $t \in [a, b]$ if we let $\Omega_t = \Omega xt \in \mathbb{R}^d \times [a, b]$ then $\partial\Omega_t$ represents the portion of Ω_t that is the boundary. If the centre nodes $\Theta \in \Omega_t$ and $\Theta = \Theta 1 \cup \Theta 2$ then $\Theta 1$ are the nodes in Ω_t excluding $\partial\Omega_t$ and $\Theta 2$ are the nodes on the boundary $\partial\Omega_t$. As with the elliptical PDE, we can show our boundary value problem as:

$$L_t = f, \text{ within } \Omega \text{ excluding } \partial\Omega_t \tag{26}$$

$$u = g, \text{ on } \partial\Omega \tag{27}$$

In this case L_t is the parabolic operator whilst f and g are the boundary functions that ensure u satisfies the PDE.

As with the elliptical problem in 24 and 25, we can now use Kansa's method to solve this system

3.8. **Spectral Methods and the Gibbs phenomena.** The Gibbs phenomenon is a peculiarity of piecewise functions:- any function that is a combination of one or more sub-functions such as a Fourier Series or in our case RBF or ATBPFs. In particular the nth combination of a piecewise function that is continuously differentiable will show persistent oscillatory characteristics in the near region of a discontinuity.These oscillatory overshoots do not reduce as n increases but do approach a finite limit. For example with the classic Fourier Series see 2 large oscillations are inherent when approximating a digital signal.

In our application, the price of a European Call Option tends toward a discontinuity around the strike price as time approaches expiry as per figure 3 so naturally there we need a method to over come it which we will take a brief look at in the next section.

3.9. **Method of Lines.** Here will briefly discuss the Method of Lines (MoL) and how it is used here, but for a complete reference see G. Meyer (2014) **??** or Y. Zhao (2017) [1]. MoL is a PDE solving technique, where all but one dimension is discretised with the last left to be continuous. In Zhao's approach, the PDE is converted into a set of ODEs which are then solved via RBF interpolation and then a Crank-Nicholson Scheme [1] pages 33-34. By careful measurement of the system's Gamma and Speed the time-stepping algorithm is employed to find a point in time

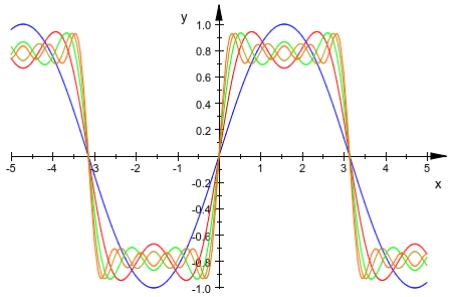

FIGURE 2. The Gibbs Phenomenon Obversed During Fourier Series Approximation of a Digital Signal
Source: https://uk.mathworks.com/help/symbolic/mupad_ug/advanced-plotting-principles-and-first-examples.html

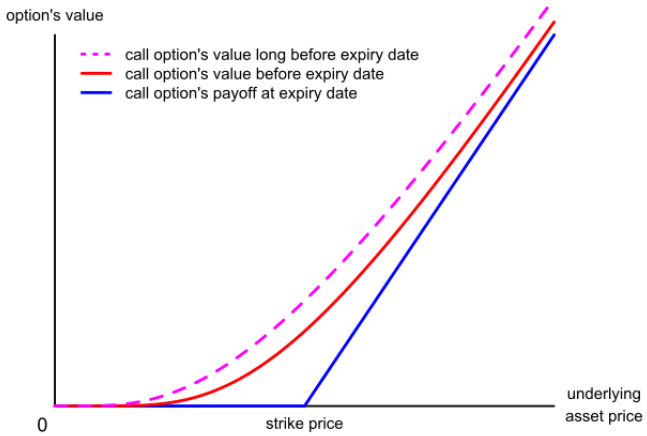

FIGURE 3. The Evolution of European Call Option's Price as Time approaches Expiry
Source: https://commons.wikimedia.org

(known as T-done) whereby the fitted curve is sufficiently smooth so as to avoid Gibb's effects. The resulting data points are then known as 'smooth initial' data throughout the rest of this paper. As a consequence, the main algorithms presented here (SiK-c and MuSik-c) are in-fact implemented in conjunction with this and other methods as shall be seen.

3.10. **Sparse Grid Collocation.** As described earlier in 3.6, it is relatively easy to use RBF interpolation in higher dimensions, but as mentioned in T. A. Driscoll and B. Fornberg.(2002) **??** the distance between nodes must be fixed in order to maintain accuracy. Therefore as the number of dimensions increases, the number of nodes (and hence the number of computations) increases proprotionally to N^d where N is the number of equi-distant points in the mesh. This phenomena is sometimes referred to as the curse of dimensionality, the concept that either the amount of computation or the space requirements of an algorithm increase exponentially with the number of dimensions.

Whilst the original ideas around Sparse grid Interpolation Kernels (SiK) originated in the Soviet Union[11][12] during the 1960s, it wasn't until Zenger's (1991) [13] work that the use of sparse grids in PDE solvers really gained traction. Zenger's method relies on the ideas set out in **??** ATRBF to save large amounts of space and computation.

A detailed explanation of sparse grids is beyond the scope of this paper, Zhao [1] (Chapter 4) gives a good description applied to SiK-c and MuSiK-c whilst Subhan [**?**] gives a more complete treatment.

To begin with, lets define the vector $l = (l_1, l_2, ...l_d)$, where $l_n = n + (d-1)$, n is the serial level and d is the number of dimensions. Then a sub-grid $G_l^(n, d)$ will contain $N^l l = 2^{l_1} + 1, ...2^{l_d}$ points in each direction. When each point point is combined into d-dimensional space the resulting number of nodes N, is:

$$N_l^{n,d} = \prod N^l = \prod_{i=1}^{d} (2^{l_i} + 1) \tag{28}$$

One sparse grid $G^{n,d}$ at level n, dimension d can be expressed as the union of all possible sub-grids $G_l^{n,d}$

$$G^{n,d} = \bigcup_{l_1 = n+(d-1)} G_l^{n,d} \tag{29}$$

Zhao gives the example on page 44 [1] of the decomposition of sparse grid $G^{4,2}$, level 4 with 2 dimensions:

$$G^{4,2} = \bigcup_{l_1 + l_2 = 5} G_{l_1,l_2}^{4,2} \tag{30}$$

Which visualised in 2 dimensions looks like figure 4

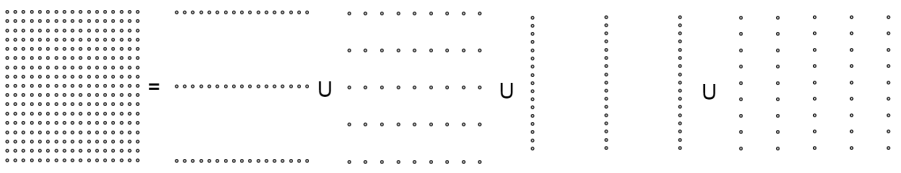

FIGURE 4. Sparse grid $G^{4,2}$ is the Union of 4 sub-grids

Zhao then speaks about removing duplicates [**?**] pages 48-49, but a union in the strictest sense ignores duplicates in the first place. Ultimately however, he identifies that the total number of nodes per full grid in SiK-c is $O(2^{dn})$ over $O(N^d)$ for a full grid. Then combined with the multi-level nature of and the space-time method in MuSiKc we end up with $O(Nlog^d(N))$

Finally, of note is the combination technique to construct the final approximation as equation31.

$$\hat{u}^{n,d}(x) = \sum_{q=0}^{d-1} (-1)^q \binom{d-1}{q} \sum_{l_1 = n+(d-1)-q} \hat{u}_l^{n-q,d}(x) \tag{31}$$

4. Implementation Details

The implementation of the algorithms described above (and detail in appendices 8.1 and 8.2), were originally written in MatLab with parrallel extensions for CPU processing and unfortunately only computes solutions to 1-dimensional (i.e. single asset) European Call Options. In order to expand this codebase (further more referred to as cMuSiK.m and cSik.m) to handle multiple dimensions and scale the available processing power across CPUs and GPUs a two step strategy was undertaken.

1 Convert the MatLab code into C++ in one go
2 Evolve parts of the C++ code into C++ CUDA as an ongoing process

As C++ is a versatile multi-paradigm language with good performance capabilities, we can be relatively confident of improving both the maintainability of the codebase whilst keeping the performance without too much micro-optimisation. Additionally C++ is sufficiently close to the native language of NVidia's CUDA that the transition from step-1 to step-2 should be relatively easy. Also the available skills base for C++ is quite high, meaning that the resulting codebase should be easier for others to understand and improve.

For the purposes of step-1 C++11 was used primarily for it's in built support for threading and the environment of Microsoft's Visual Studio 2017 due to the familiarity of the author and it's availability as a free tool-set. Care was taken to ensure no platform specific libraries were used such that compilers other than Microsoft's implementation should easily be able to compile the codebase given an appropriate build system. The resulting Visual Studio Solutions, a mixed C++ and CUDA implementation (refered to as SparseGridCollocation.sln) and a pure C++ implementation (referred to as SparseGridCollocationCPP.sln) are the results of this conversion process.

In both solutions the code is generally arranged within a namespace heirachy and seperated into sub-projects:

Organisation:	Leicester
Research area:	SparseGridCollocation
Program sub system:	ProjectSpecific

Specific functions and classes within the solution will be referred to by the Namespace::Class::Function convention i.e,

- Leicester::SparseGridCollocation::Algorithmn::MuSiKc() refers to the main entry point of the MuSiKc implementation, whilst
- Leicester::Common::printMatrix() refers to a particular helper function that prints the contents of a matrix to a string.

Each subproject will compile into a separate dynamically linked library (.dll) or an executable program (.exe). For C++ specific code (i.e. projects containing files with extensions .cpp and .h) the msvc compiler will perform the compilation and linking whilst CUDA specific projects (extensions .h or .cuh and .cu) are handled by the NVidia nvcc compiler. Details of both these compilers and their respective software development kits (SDKs) can be found within the appendix.

4.1. Reverse Engineering via Unit Testing.

A common method known within the software engineering community to reverse engineer computer programs who's

source code is unavailable is the process of black-box unit testing[15]. This is the process of treating the program as a 'black-box' where the true functionality of a particular program can be revealed by carefully selecting the input to the program and measuring the resulting out puts.

For example if we input the full range of integers into a black-box and observe the results to be squares of the input then we can be reasonably confident that the program implements $F(x) = x- > x^2$

Whilst in our case the source code is available, this technique can still provide us with a method of measuring and detecting any difference in functionality between our new C++ code and that of the original MatLab code.

4.2. **Eigen API and MatLab comparison.** Eigen8.5 is a useful library that styles itself as a 'C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms.' which is open source and distributed under an MPL license. What makes this particular library stand out from others is its coverage of MatLab functions. In particular Zhao makes use of the following MatLab functionality that Eigen also replicates:

- Linspace - create a linearly spaced vector within a range for a certain number of points
- zeros, ones and fill
- size
- Column & Row addressing
- row wise and column wise sum
- Meshgrid
- LU decomposition
- LU Solver

A full list of MatLab emulated functionality can be found within the documentation**????**.

4.3. **Column Major Format.** There are two main methods for representing a Matrix within computer memory row-major and column-major formats. With both, each element within the matrix is mapped to an element within contiguous memory. The difference between them however is the exact mapping between those rows and columns and the consecutive elements within memory. It is therefore worth descibing in some detail the exact differences and the convention used here.

An array of values (for example the values 1,2,3...n) are normally stored as consecutively at equally spaced addresses within computer memory, this is known as contiguous memory. If the first element is stored at address A and the size of each value is x, then for element n within an array of length N, the nth element is stored at A + (x * n).

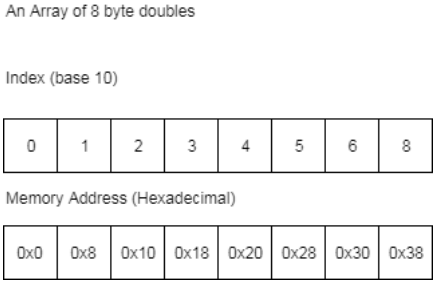

FIGURE 5. Contiguous Array

In row-major format each row is appended to the array, with each row element at consecutive locations

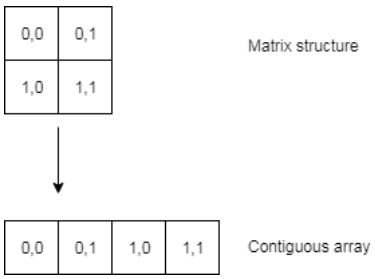

FIGURE 6. Row-Major Format

In column- major format each column element is at a consecutive location, while each column is proceedingly appended to the end of the array

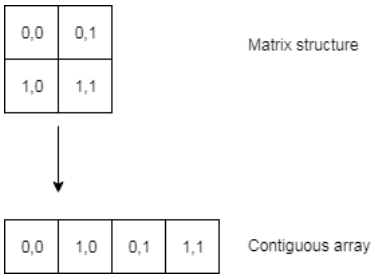

FIGURE 7. Column-Major Format

The convention used within this project is column-major format, which is the same method as both the Eigen library and that of MatLab. Unfortunately the array construct in C and C++ is simply a pointer to the first element and no further information is provided. Consequently unless the dimensions of an array are known at compile time, for example:

```
int m[] = int[10];
```

or unless the programmer stores the dimensions of a dynamic array:

```
int rows =10;\\
int cols = 10;\\
int *m = int[rows*cols];\\
```

it is impossible to reconstruct the array or matrix from contiguous memory which has big implications that we will see in section **??**

4.4. **Expressions and Lazy Evaluation.** Another advantage of Eigen is a useful optimisation for handling wasteful processing called expression trees, which as a concept are similar to those seen in C# LINQ or SQL. Designed to reduce the number of repeated parsing of a Matrix or array by the programer, Eigen 'records' at compile time all operations performed on a particular matrix and constructs an 'expression' representing the aggregate.Eigen then generates the optimal code to perform the aggregate operation on the matrix in as few loops as possible. Eigen will often do this as late as possible and when done so at actual run-time is known as Lazy Evaluation.

For example, say the programer wanted to add a constant to each element of a Matrix and then later on in the program square each element and so they might write:

```
int rows = 10;
int cols = 10;
MatrixXd m = MatrixXd::Ones(rows, cols);
int c = 1;

for (int i = 0; i < rows; i++)
    for (int j = 0; j < cols; j++)
    {
        m(i, j) += c;
    }

for (int i = 0; i < rows; i++)
    for (int j = 0; j < cols; j++)
    {
        m(i, j) *= m(i, j);
    }
```

FIGURE 8. Inefficient Matrix Parsing

In this example it is easy to see that each element of the matrix 'm' is visited by the program twice, but this is obviously wasteful when one considers the following:

```
int rows = 10;
int cols = 10;
MatrixXd m = MatrixXd::Ones(rows, cols);

int c = 1;
for (int i = 0; i < rows; i++)
    for (int j = 0; j < cols; j++)
    {
        int a = m(i, j) + c;
        m(i, j) = a * a;
    }
```

FIGURE 9. Efficient Matrix Parsing

Where a more efficient programmer will realise that each element of 'm' only needs to be visited once by the program. Eigen's expressions perform the same kind of optimisation when working with Eigen's Matrix template classes, freeing the programmer to concentrate on writing functioning code rather than laboriously optimising their code.

4.5. **Intel Math Kernel Library.** All the algorithms studied within this paper make use of some linear algebra and MuSiK-c in particular uses vector-vector,

vector-matrix, matrix-matrix (i.e. BLAS levels 1-3) and LAPACK routines for decomposition and solving linear systems.

Whilst it is possible to use Eigen without any further dependency, it is recommend that instead of using it's inbuilt support for BLAS and LAPACK it is best to use a platform specific provider. In our case, we have chosen Intel Math Kernel Library[16] (MKL) which is optimised for Intel based chips[17] and their SSE[19], SSE2 and SSE3 instructions sets for Single Instruction, Multiple Data[18] (SIMD)

Briefly, SIMD allows the processor to stream multiple datasets onto a CPU core in parallel during one instruction. A core is in essence an independent processing unit and modern CPUs are designed to have several of these all on the same chip. What this means is that the same operation can be performed at the same time on several pieces of data simultaneously. Without SIMD that single instruction would need to be executed in serial on each data element individually, which is clearly a vast improvement.

This is particularly useful when used in conjunction with multiple threads operating on a multi-core CPU. Each thread of execution operating on a per core basis can process large data sets themselves in parallel giving us effectively two layers of parallelism.

4.6. **Parallelisation with C++ Threading and CUDA.** Whilst one of the main goals of this paper is improve the performance of MuSiK-c via parallelisation, it is important to note that not all algorithmns are can take advantage of such improvements. A program is said to have parallelism or be parallelisable if all or a number of it's operations can occur simultaneously without changing the overall functionality of said program. Otherwise, the algorithm is said to exhibit Sequential behavior.

Taking MuSiK-c as an example (see 8.2), we see that some sections are inherently sequential (e.g. the need to process each level using the state of prior levels) and some areas can be run in parallel (e.g the interpolation via ATBPFs or RBFs). As multi-threading on the CPU is a well known topic, we will skip this and describe how GPU programing with Nvidia's CUDA achieves massive parallelisation and how CUDALib.vcxproj implements some simple parallel routines.

4.7. **NVidia GPU Hardware architecture.** GPU hardware differs some what from that of CPU architecture figure 10 shows how a GPU packs far more cores on to chip than does a CPU. Cores are organised into Streaming Multiprocessors (SM) aka a Warp the GPU equivalent of a CPU's SIMD enabling parallelisation of data flow into and out of the GPU.

Whilst it might seem from this that GPUs are far superior to CPUs where the are lacking is in the amount of RAM the GPU cores can access. For example the reference hardware in 8.6 is a current (as of 2017) GPU containing 2Gb of memory, whereas the CPU (a 2011 vintage) can access the entire 16 Gb of system RAM with ease. As such, to make effective use a GPU's computing power, the CPU must always be busy sending data down the line to the GPU device.

CUDA's threading model is shown in figure 11 and like all concurrent execution models, the unit of execution is known as a thread but in CUDA threads are also organised using Blocks and Grids**??**cuThrd).

- Thread: A unit of sequential execution, in CUDA all threads execute the same sequential program in parallel.

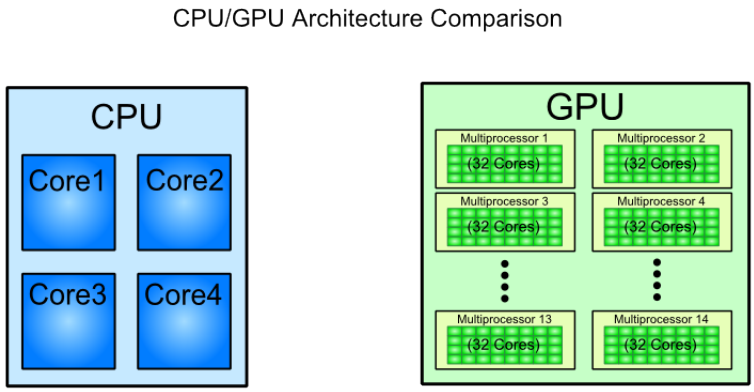

FIGURE 10. CPU vs GPU Architecture
Source: http://blog.goldenhelix.com/grudy/video-graphics-and-genomics-a-real-game-changer/

- Thread Block: A group of threads which all execute on the same streaming multiprocessor. Threads within a block can synchronise with each other and exchange data.
- Grid: A group of Thread Blocks where blocks execute across multiple SMs, can't synchronise and communication between is expensive.

The CUDA run-time API provides a thread object that allows the programmer to identify the x-y-z 3-dimensional coordinate of the thread within it's block as well as the same data for the block within the grid.

Initally, it seems that CUDA is designed specifically for our purposes: parsing large NxM matrices in parallel. As we can keep track of the Thread's x and y indices within the larger grid and block system we launch a grid with enough blocks and threads (over 10,000 concurrently on the reference hardware alone), to process each element of the matrix in parallel. Likewise as the most efficient method is to use blocks of 32x32 threads we can launch more threads than we need but let some 'out of index' threads process nothing.

In general a CUDA program's functions are organised into 3 main types**??**cuPrg based on whether they are executed on the host-side (the CPU) or the device-side (the GPU):

- Kernel: What is termed as the host-side program i.e. operates as a normal C++ program on the CPU able to address system memory and call the Cuda Runtime API
- Global Functions: A device-side function that can be called from the Kernel but also from other device-side functions. These functions must be executed by defining grids, blocks and threads that will execute the function concurrently.
- Device Functions: Functions that can only be called by device side code and as such themselves run on the device. As with Global Functions must be executed with a declared set of threads.

The CUDA runtime is rather limited, in that useful libraries such as the C++ Standard Template Library (STL)**??** or Eigen are not usable on the GPU. Furthermore the run-time only supports the copying of data from host to device (or device to host) via the use of pointers. Both of these drawbacks present significant problems to the programmer. A matrix therefore must be represented by a pointer

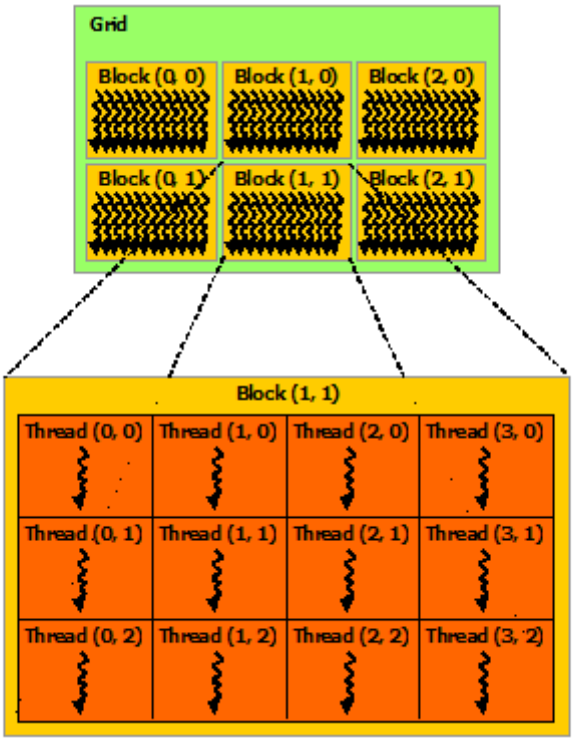

FIGURE 11. CUDA threading model
Source: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.htm

construct (hence our explanation or column-major format in section 4.3) in order to be loaded by the GPU and passed across functions. Likewise once copied back to the host, the pointer array must be mapped back to a matrix structure. For MuSiK-c this means that our dynamically sized matrices are difficult to handle correctly in CUDA unless we pass the dimension sizes around with them. This also complicates (very significantly) the implementation of our grid interpolations as we must keep track of different indices pointing to different elements in our Test Nodes and our Central Nodes.

To add further complexity to the mix, imagine you wanted to execute a grid with n blocks of m threads but each thread itself had to spawn another grid of p blocks and q threads. CUDA can do this as well via what it calls *Dynamic Parallelism*?? an example of which can be seen in CudaLib.vcxproj where $CudaLib :: Gaussian2d\_CUDA$ launches $CudaLib :: Gaussian2d2\_CUDA$. Dynamic Parallelism is important for applications such as MuSiK-c where there are at least three layers that can be parallelised i.e. ShapeLambda and two levels of RBF Interpolation . The big drawback is that without multiple graphics cards installed on your system, it is not possible to debug these kinds of routines.

As a result of the difficulty in indexing dynamically sized matrices in contiguous memory and that the reference hardware only included one CUDA capable GPU, CudaLib works reasonably well for 1 dimensional European Call Options at low levels. However not so for the larger matrices encountered at level 7 and above or for Basket options.

4.8. **CUDA Thrust Library.** CUDA Thrust**??** is an abstraction over the main CUDA programming model. It in effect allows the user to define their matrices in an STL-like vector container, define a special function called a 'functor' and the thrust library will automatically copy the matrix to the GPU and execute the functor against every element in the matrix. What is especially useful in our case is that the functors are stateful, such that they are defined as a C Struct type allowing us to know within the function body the dimensions of the various matrices that we are processing as well as any further supporting data we may wish to access.

5. C

omputational Experiments The following experiments are designed to verify the correctness of the new codebase versus that of the legacy MatLab as well as indicate relative performance differences. Each experiment is run twice, once for the pure C++ version and then once for the mixed CUDA C++ version. Full output of each experiment is shown in the appendices.

5.1. **C++ Experiments.** Compiled from Usng SparseGridCollocationCpp.sln, Experiments.vcxproj
Run using the command Experiments.exe [experiment number]
where [experiment number] corresponds to the sub section number below:

5.2. **Experiment 1.** Purpose: Verification of Method of Lines 1 Dimension in C++ vs MatLab (MOL to find an earlier time) To_get_ini.m

Viewing the graphical comparison in figure 12 of MatLab versus our one dimensional Method of Lines implementation shows an exact match, which suggests our implementation is functionally equivalent. We will therefore continue to use this new code to generate the smooth initial data used in the main algorithms.

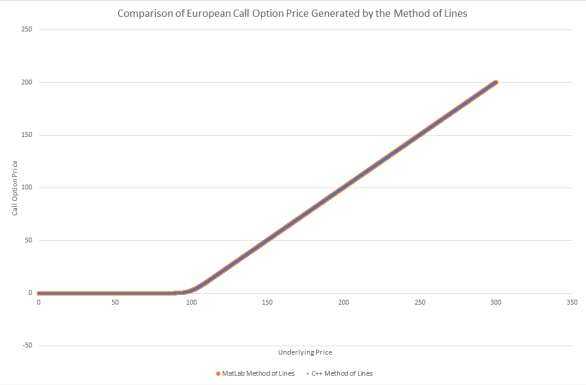

FIGURE 12.

5.3. **Experiment 2.** Purpose: Verification of SiK-c 1 Dimension in C++ vs MatLab (sparse grid collocation) cSIK.m

In this case the graph 13 clearly shows a difference in the implementations of SiK-c in C++ and that of the original MatLab code. From **??** we would expect to see far smaller Root Mean Squared (RMS) errors than we actually do.

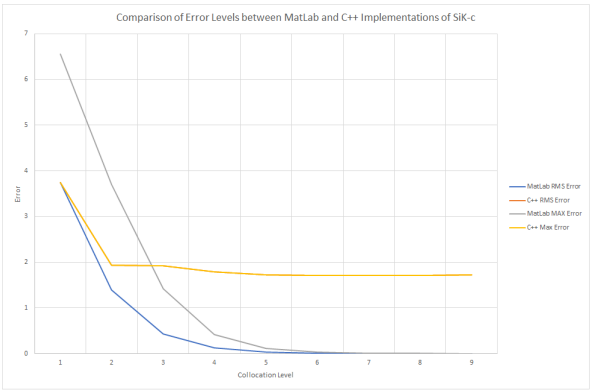

FIGURE 13.

5.4. **Experiment 3.** Purpose: Verification of MuSiK-c 1 Dimension C++ vs MatLab (sparse grid collocation) cMuSIK.m

Whilst the calculated points may not match, the graph 14 shows that even at logarithmic scale the RMS and Maximum errors are almost exact and trending in the same direction, suggesting the C++ version is equivalent

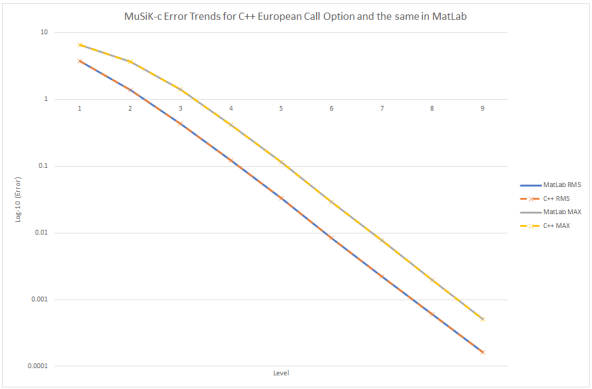

FIGURE 14.

Measuring performance, we can see that C++ outperforms MatLab at lower collocation levels, in this case. This is perhaps partially due to the constant time required for MatLab to launch it's parallel pool. However as no thread pooling is used in the C++ implementation then this is a fair test.

5.5. **Experiment 4.** Purpose: Verification of Method of Lines N Dimensions, 1 asset European Basket Call Option in C++ vs MatLab (MoL to find an earlier time) To_get_ini.m

Now turning to the multi-dimensional versions of our C++ code, we choose a Basket Option with 1-underlying asset, whose pay off is related to the arithmetic mean of the underlying. This means that a 1-asset Basket Call Option should have the same price as a European Call Option on the same asset where the strike prices are the same.

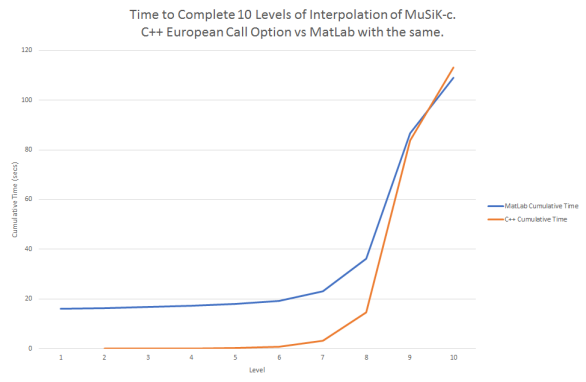

FIGURE 15.

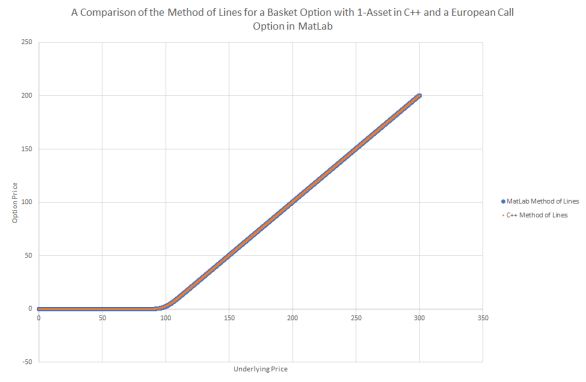

FIGURE 16.

Indeed, for our Method Lines in N-Dimensions seems to be equivalent to the same MatLab procedure with same inputs. Usefully we can now use this the output here to feed the other algoirthmns in multiple dimensions.

5.6. **Experiment 5.** Purpose: Verification of MuSiK-c N Dimensions, 1 asset European Basket Call Option in C++ vs MatLab (sparse grid collocation) cMuSIK.m

Using the output of the last experiment we now feed the smooth initial data to the main MuSiK-c algorithmn and compare it to MatLab's output. Graph 17 shows a continued good match even at logarithmic scale, despite a small (but negligible here) drift upwards in the error as collocation levels increase.

As we saw previously in **??** and now in graph **??** C++ also outperforms MatLab at lower collocation levels even though it must work a little harder to cater for multiple dimensions.

5.7. **Experiment 6.** Purpose: Verification of Method of Lines N Dimensions, 2 asset European Basket Call Option in C++ vs QuantLib Montecarlo 2-Asset

We are sceptical of how well MoL will scale with extra dimensions and whilst we have already done the work to parallelise RBF/ATBPFs in the main MuSiK-c algorithm the process for finding T-done is inherently sequential and hence slow. Instead, here we will begin testing a more scalable method of generating smooth

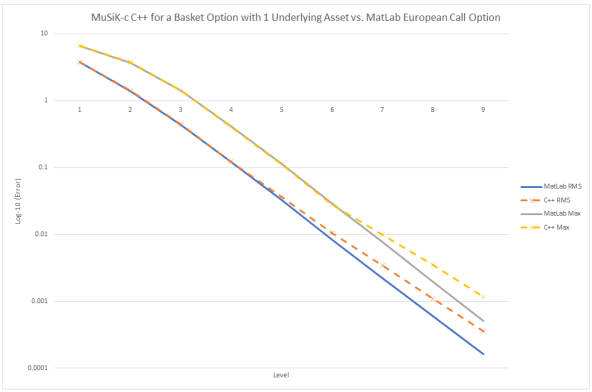

FIGURE 17.

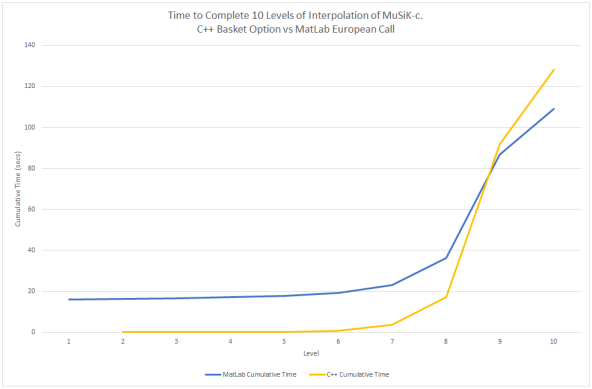

FIGURE 18.

initial data and make the *big assumption* that T-done for 2 assets is the same as T-done for N-assets: QuantLib's[**?**] implentation of a Montecarlo algorithm for basket options.

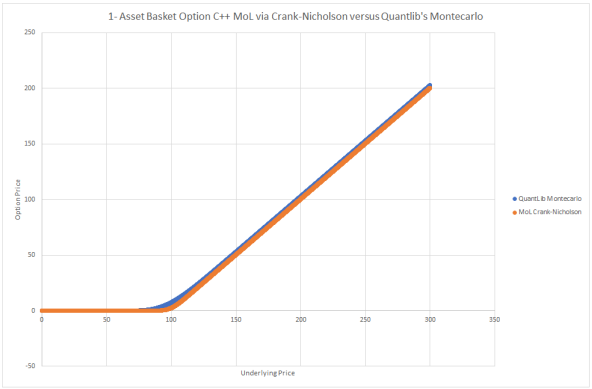

FIGURE 19.

Graph 19, shows that (as we might have expected) Montecarlo is very close in terms of accuracy to MoL, at least beyond the near region of the strike. Both MuSiK-c and SiK-c rely on the smooth initial data generated here to supplement

their calculations close to the boundary represented by T-done, so the selection of T-done using Montecarlo may well have a big effect later on.

5.8. **Experiment 7.** Purpose: Verification of MuSiK-c N Dimensions, 2 asset European Basket Call Option with QuantLib MC as method of choice in C++

Extending our Basket Option to 2 Underlying Assets presents a real challenge to our method. Graph 20 shows errors only increasing until we reach the cap of the number of levels we chose to run. A cap of 6 levels is made, due to the process using up all available system memory from the 7th level onwards. At this time, we are unsure whether the difference in error is due to our choice of T-done, whether we're not generating the N-matrix (and hence our shape parameters) correctly, there is a bug in the code or there is an inherent problem with MuSiK-c. We could invest in more memory for the reference system or find some memory management improvements to reduce the algorithm's footprint.

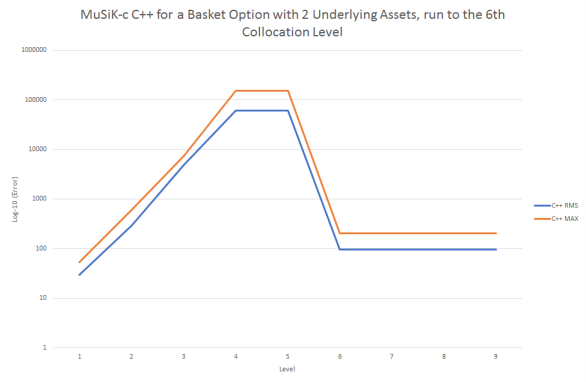

FIGURE 20.

Performance figures in 21, prove to be disappointing.

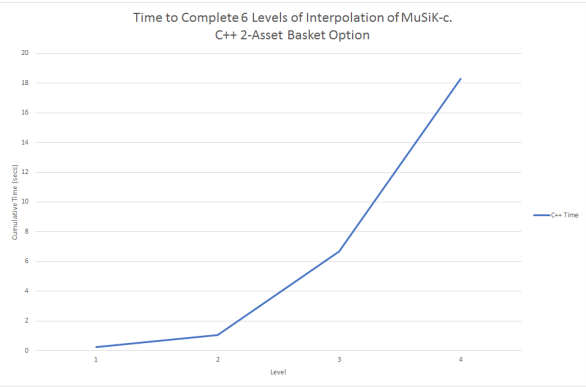

FIGURE 21.

5.9. **CUDA C++ Experiments.** Compiled from Usng SparseGridCollocation.sln, Experiments.vcxproj
Run using the command Experiments.exe [experiment number], where [experiment number] corresponds to the sub section number below:

5.10. **Experiment 3.** Purpose: Verification of MuSiK-c 1 Dimension in C++ CUDA vs MatLab (sparse grid collocation) cMuSIK.m

We now benchmark our 1-dimensional CUDA C++ implementation with that of the original code. Graph **??** shows an exact match between both, so we can verify that this implementation is correct.

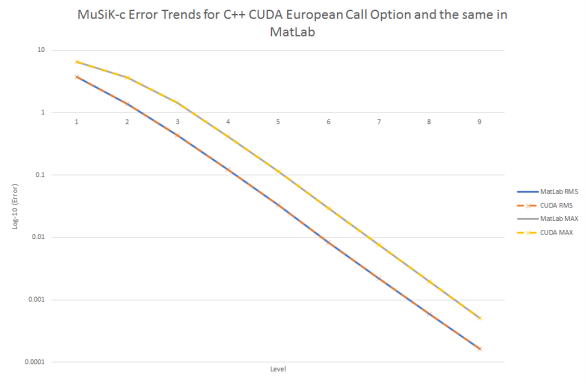

FIGURE 22.

However when we explore performance in the graph 23, we see that the CUDA implementation is significantly slower at higher levels and marginally so at lower ones. Whilst there are 'warm up' costs for launching kernels on the GPU (which perhaps explain the constant difference below level 8), these do not account for the poor performance at higher level.

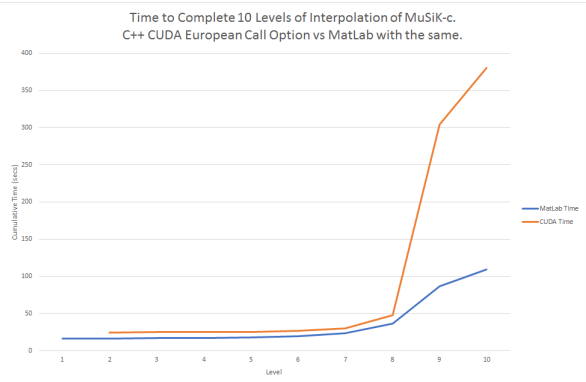

FIGURE 23.

5.11. **Experiment 5.** Purpose: Verification of MuSiK-c N Dimensions, 1 asset European Basket Call Option in C++ CUDA vs MatLab (sparse grid collocation) cMuSIK.m

As with the equivalent experiment in pure C++ 5.4, graph **??**fig:cuExp5e reveals that our CUDA implementation is in good agreement with the original code. Again we see a slight drift in both RMS and MAX errors once we reach higher levels, which suggests a systematic difference in the way we are calculating the derivatives of the ATBPFs.

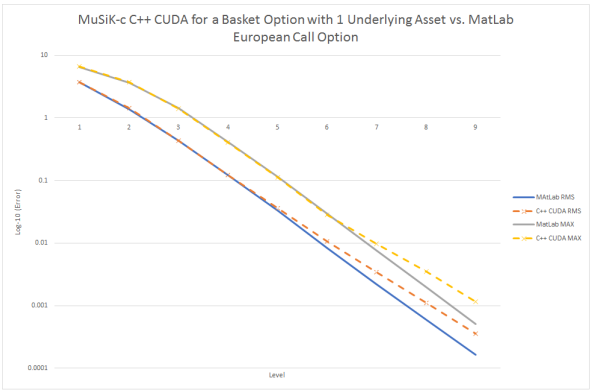

FIGURE 24.

Performance wise, its a similar story to the prior experiment, however CUDA seems to out perform at the lower levels before this advantage receeds at high levels. We know that we've not fully parallelised the PDE-reconstruction (which is used proportionally more at higher levels), in this version of the code so perhaps by doing so we will gain a better advantage.

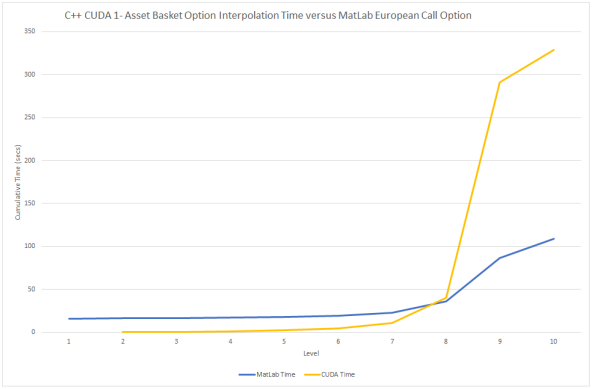

FIGURE 25.

5.12. **Experiment 7.** Purpose: Verification of MuSiK-c N Dimensions, 2 asset European Basket Call Option with QuantLib MC as method of choice in CUDA

As with the corresponding pure C++ experiment 5.8, the number of levels are capped to 6 due to the same problems and in addition GPU memory consumption contraints. Graph 26, shows that whilst there is a slight improvement in accuracy at the lowest levels some problem with our program causes the errors to quickly mount up beyond level 2.

Again, the performance figures shown in **??** are disappointing worse so than with 1-asset. However it may be worth investigating if theses only scale linearly with dimensions.

5.13. **Profiling Results.** Both versions of the algorithm have been profiled to gain insight into potential bottle necks, for C++ see**??** and for the GPU profile of CUDA see**??** Of the 5 hottest functions within our C++ version, 4 are library code (Intel MKL) and the last is our implementation or RBF interpolation. This suggests that we are correct in our assumption that this part of the software is perhaps best run

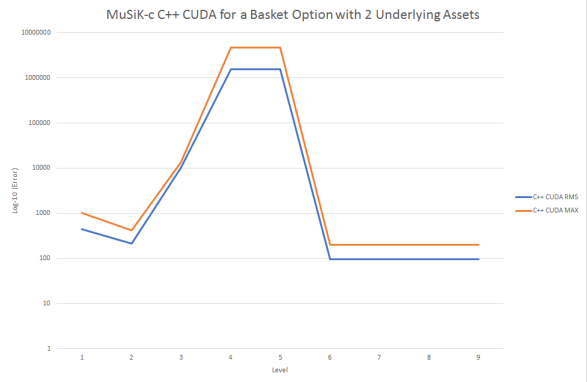

FIGURE 26.

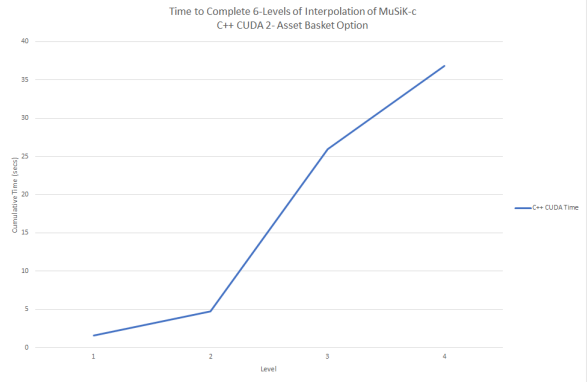

FIGURE 27.

on the GPU. The CUDA profiler is a little more difficult to read, however the main item is concern is that we seem to be mainly using the default stream processor and the other streams for little else. This potentially is the cause of the helpful warnings at the bottom concerning low through put, memcopy efficiency and low compute.

6. Conclusions

We have proven that both our C++ and C++ CUDA implementations are functionally equivalent to the original MatLab code in the 1-Asset case for European Call Options developed by Zhao for his PhD. Thesis [1]. Further, we have extended his algorithm to cater for Basket Options and other higher-dimensional instruments. Unfortunately we can only prove the correctness of this approach using 1-Asset instruments.

In terms of hardware and software scalability we have learnt that C++ in conjunction with Eigen and Intel MKL allow us to scale the number of CPUs and size of system Memory reasonably well. However we should note, that a more in depth forensic approach to optimisation should be undertaken before claiming this. In terms of heterogenous programing across both CPU and GPU, we have learnt that significantly more care needs to be taken using and optimising algorithms for NVidia CUDA. Whilst our implemntation does use CUDA, so far we've been unable to push (and remove) enough data from the GPU memory at fast enough rates to really push the reference hardware to it's limits.

7. Further Work

As was noted in section 3.4 shape parameters have a significant bearing on the out-come of all RBF interpolants. Whilst it seems that we are using close to the correct method of generating both them and the sub-grids with 1-Asset, this is perhaps not the case with more than 1. There are perhaps 2 approaches to discovering whether this is the case. Firstly we could return to the theoretical background and re-affirm whether we are adhering to the ideas there. Secondly, we could build some automated learning method to scan through a range of shape parameters and discover the optimal values.

We are also unconvinced by our method of choice for generating smooth initial data. As we have seen, the Method of lines does not seem to scale well as dimensions increase. Further more, we have not investigated our alternative Montecarlo approach sufficiently to eliminate any extra problems it might bring to MuSiK-c. In particular, it lacks the ability to find a suitable T-done for RBF interpolation so requires the user to make an 'educated guess' as to where to start MuSiK-c. If this is done incorrectly, the method will encounter Gibb's phenomenon and so lose accuracy close to dis-continuities.

In terms of software, there is plenty of optimisation to be done both in terms of memory footprint (and through put), thread pooling, efficient algorithms and further parallelisation. Each of these areas would need significant attention should this implementation ever be considered for production use. In addition, the unit and regression testing coverage is neither large nor automated in an efficient manner. Addressing that would significantly improve the ability of other code contributors work on the project.

In terms of hardware, we have barely scratched the surface. Whilst this solution has been deployed and run (on occasion with a virtual machine on Google Compute, it has only been run on very limited cloud hardware. GPU enabled servers are available to use for the professional subscriber and in the near future NVidia will launch their on GPU based Grid-Cloud hybrid**??**. So the project is in a good position to take advantage of new innovations in this space.

8. Appendices

8.1. **SiK-c Details.** The basic Sparse Grid Collocated Interpolation Kernel (SiK-c) is shown in figure 28 is our basic algorithmn which once expanded to multiple levels of collocation becomes MuSiK-c.

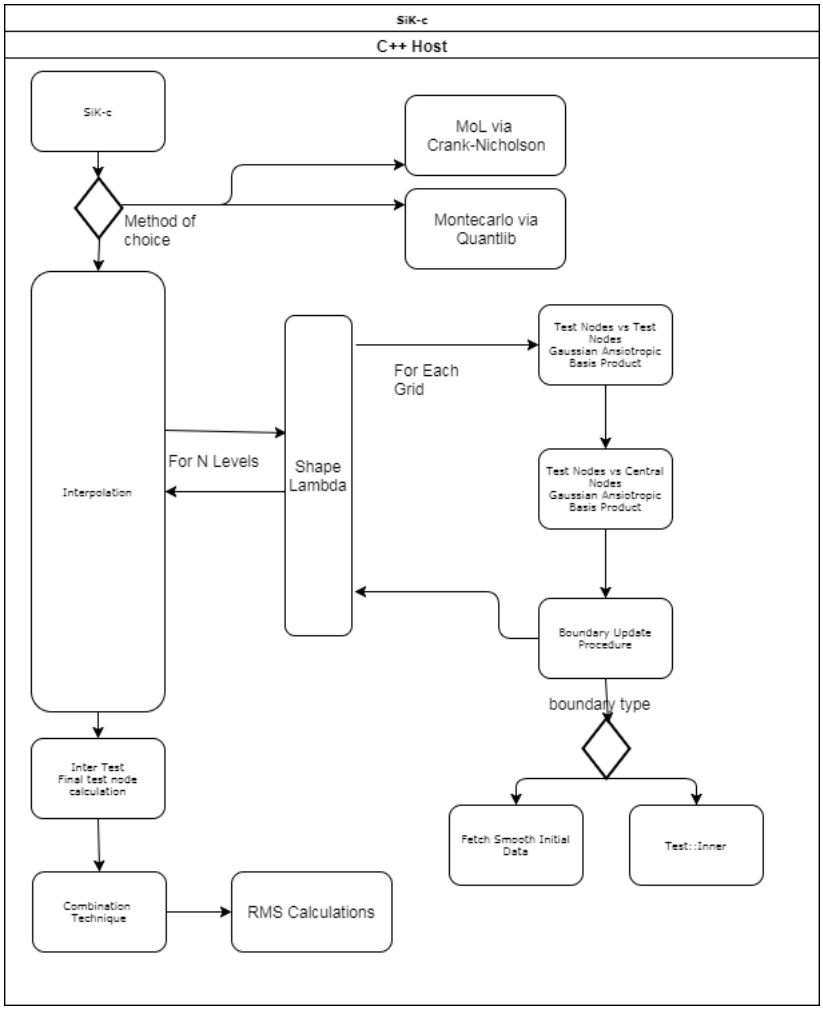

Figure 28. The SiK-c Algorithmn

8.2. **MuSik-c Details.** MuSiKc on the other hand use multi-level co-located interpolation, where the results of each level are re-used as inputs for the next. As such, MuSiK-c is an inherently sequential evolution of SiKc.
The following diagram29 shows the processing flow of the main algorithm and it's differences to SiK-c.

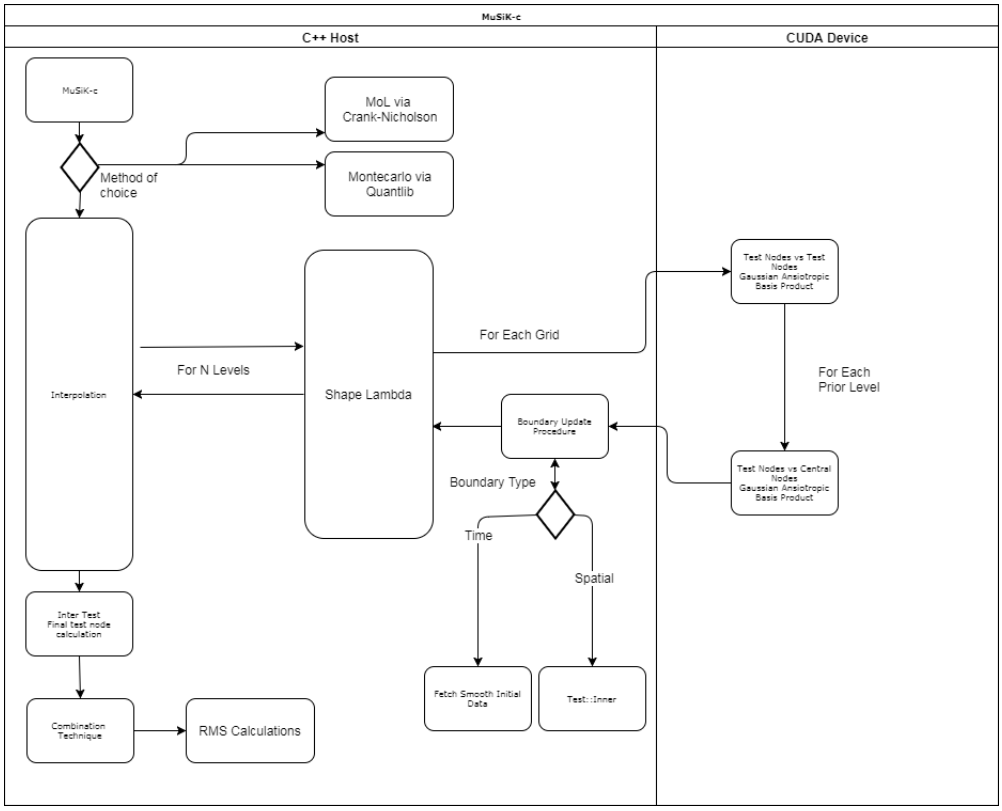

FIGURE 29. The MuSiK-c Algorithmn

8.3. **D. Navigating from Theory to Source Code.** Theoretical points of interest and their C++ implementations

The methods of choice for generating smooth initial data.

- Method of Lines via Crank-Nicholson Leicester::SparseGridCollocation::MoL::MethodOfLines
- Montecarlo Leicester::SparseGridCollocation::Montecarlo::BasketOption

The generation of N- matrix 28 and the generation of Test Nodes for each level is implemented within:

- Leicester::SparseGridCollocation::TestNodes::GenerateTestNodes
- Leicester::SparseGridCollocation::Interpolation::primeNMatrix
- Leicester::SparseGridCollocation::Interpolation::subnumber

Ansiotropic Tensor Based Product Interpolation Using Gaussian RBFs

- C++ Leicester::SparseGridCollocation::RBF::Gaussian2D
- C++ CUDA Leicester::ThrustLib::GaussianNd1::GaussianNd1
- General Interpolation and Boundary Update Procedure Leicester::SparseGridCollocation::Interpolati

8.4. **Further Information on Automated Testing.** SparseGridCollocation.sln and MuSiKc.m include a number of methods and frameworks in order to isolate particular modules of functionality some of which are:

- Leicester::SparseGridCollocation::checkMatrix() - Of which there are 3 versions, allowing the user of the function to compare with configurable precision, the sameness of 2 matrices.
- Leicester::Common::Utility::printMatrix() - Including several variants that will write the contents of a matrix to a string in various formats prior to inserting into cout and printing to the console
- Leicester::Common::Utility::saveMatrix() - Save the contents of a matrix to a human readable text file
- Leicester::Common::Utility::saveMatrixB() - Save the contents of a matrix to a binary file so as not to lose precision but the dimensions of the matrix are not saved.
- WriteAllToFile.m - Save the contents of a matrix to a binary file.
- TestHarness.vcxproj - Contains a simple console application to use as a workbench for running various functions and comparing their output.
- Microsoft Test Frame work - The UnitTest.csproj contains a number of unit tests that when executed within Visual Studio will run parts of or the entire algorithmn of choice with a standard input and check the actual output against an expected output

8.5. **E. Code, Repositories and Dependencies.** All source code for this paper can be accessed via Git or Subversion from GitHub.com via:
`https://github.com/NicholasWilton/SparseGridCollocation`

Dependency	Version	URL	Description
Eigen	3.34	http://eigen.tuxfamily.org	General purpose C++ Matrix and Linear Algebra Library
Intel MKL	2017.4.210	https://software.intel.com/en-us/mkl	SIMD parallel processing improvements for BLAS and LAPACK algorithmns on Intel processors
NVidia CUDA Toolkit	8	https://developer.nvidia.com/cuda-downloads	GPU-SDK for NVidia graphics cards

8.6. **Reference Hardware.** The following system was used to both develop and test the new software.

Component	Specification
Operating System	Windows 7 Professional 64-bit (6.1, Build 7601) Service Pack 1
CPU	Intel(R) Core(TM) i5-2500 CPU @ 3.30GHz (4 CPUs), 3.3GHz
Memory	16384MB RAM
Graphics Card	NVIDIA GeForce GTX 1050 2GB DDR (640 cores, organised into 5 Streaming Multiprocessors with 2048 Threads per Processor)
Hard Drive	488.2 GB SSD Drive

8.7. **Index of Experimental Results.**

C++ Experiment Number	Relative Source Location
1	(root.dir)/SparseGridCollocationCpp/Experiments/1
2	(root.dir)/SparseGridCollocationCpp/Experiments/2
3	(root.dir)/SparseGridCollocationCpp/Experiments/3
4	(root.dir)/SparseGridCollocationCpp/Experiments/4
5	(root.dir)/SparseGridCollocationCpp/Experiments/5
6	(root.dir)/SparseGridCollocationCpp/Experiments/6
7	(root.dir)/SparseGridCollocationCpp/Experiments/7

C++ CUDA Experiment Number	Relative Source Location
3	(root.dir)/SparseGridCollocation/Experiments/3
5	(root.dir)/SparseGridCollocation/Experiments/5
7	(root.dir)/SparseGridCollocation/Experiments/7

8.8. **Profiling Results.**

FIGURE 30. Profiling the C++ Process with CodeXL

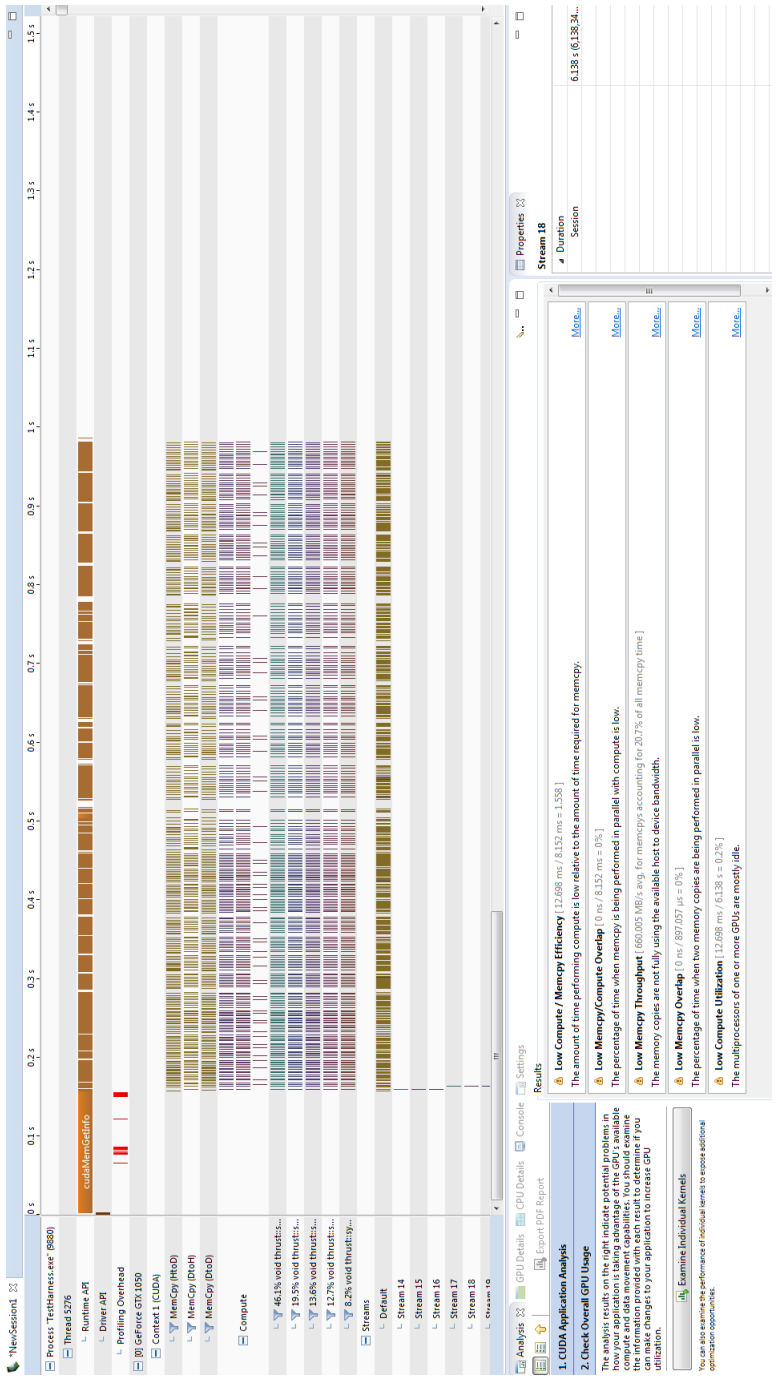

FIGURE 31. Profiling the C++ CUDA Process with NVidia Visual Profiler

References

[1] Zhao, Yangzhang (2017), *Multilevel sparse grid kernels collocation with radial basis functions for elliptic and parabolic problems*, Ph.D. Thesis, University of Leicester, 2017.

[2] Subhan, F. (2011), *Multilevel sparse kernel-based interpolation*, Ph.D. Thesis, University of Leicester, 2011.

[3] Georgoulis, E.h., Levesley, J., Subhan, F. (2013), *Multilevel sparse kernel-based interpolation*, SIAM Journal on Scientific Computing, 2013, 35(2):A815-A831.

[4] Myers, D.e.,De Iaco, S.,Posa, D.,De Cesare, L. (2002), *Space-time radial basis functions*, Computers and Mathematics with Applications 2002 43(3):539-549

[5] E. J. Kansa. (1990), *Multiquadrics - a scattered data approximation scheme with applications to computational fluid-dynamics - I.*, Computers and Mathematics with Applications, 19(8-9):127–145, 1990.

[6] E. J. Kansa. (1990), *Multiquadrics - a scattered data approximation scheme with applications to computational fluid-dynamics - II.*, Computers and Mathematics with Applications, 19(8-9):147–161, 1990.

[7] R. Franke. (1982), *Scattered Data Interpolation: Tests of Some Method*, Mathematics of Computation. 38(157):181-200, 1982

[8] T. A. Driscoll and B. Fornberg.(2002), *Interpolation in the limit of increasingly flat radail basis functions.*, Computers and Mathematics with Applications, 43(3):413–422, 2002.

[9] B. Fornberg, E. Larsson, and N. Flyer. *Stable computation with Gaussian radial basis functions.*, SIAM Journal on Scientific Computing, 33(2):869–892, 2011.

[10] A. Pena. (2005), *Option pricing with radial basis functions: a tutorial.*, Technical report, Wilmott Magazine, 2005.

[11] K. I. Babenko. (1960), *Approximation by trigonometric polynomials in a certain class of periodic functions of several variables*, Soviet Mathematics Doklady, 1:672–675, 1960.

[12] S. A. Smolyak. (1963), *Quadrature and interpolation of formulas for tensor product of certain classes of functions.*, Soviet Mathematics Doklady, 4:240–243, 1963.

[13] C. Zenger. (1991), *Parallel algorithms for partial differential equations.*, Notes Numerical Fluid Mechanics Vol 31 pages 241-251. Vieweg, Braunschweig, 1991.

[14] B. Fornberg, N. Flyer. (2011), *The Gibbs phenomenon for radial basis functions.* The Gibbs Phenomenon in Various Representations and Applications. A. Jerri, Sampling Publishing, Potsdam, NY, (2011), Chapter 6, 201-224 @bookmeyer2014time, title=The Time-Discrete Method of Lines for Options and Bonds: A PDE Approach, author=Meyer, G.H., isbn=9789814619691, series=Peking University-World Scientific Advance Physics Series, url=https://books.google.co.uk/books?id=bxu3CgAAQBAJ, year=2014, publisher=World Scientific Publishing Company

[15] K. Beck. (2002), *Test Driven Development: By Example.*, Addison-Wesley, 2002.

[16] Intel (2017), *Developer Reference for Intel Math Kernel Library*, https://software.intel.com/en-us/mkl-developer-reference-c

[17] Intel (2017), *Intel Streaming Extensions Technology*, https://www.intel.co.uk/content/www/uk/en/support/processors/00000

[18] Wikipedia (2017), *SIMD* https://en.wikipedia.org/wiki/SIMD

[19] Wikipedia (2017), *Streaming SIMD Extensions* https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions

[20] Nvidia (2017), *Memory Optimizations, CUDA C Best Practices Guide* http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.htm

[21] Eigen (2017), *Eigen Quick Reference* http://eigen.tuxfamily.org/dox-devel/AsciiQuickReference.txt

[22] Eigen (2017), *Eigen Documentation* http://eigen.tuxfamily.org/dox/index.html
http://docs.nvidia.com/cuda/

[23] NVidia (2017), *CUDA Programming Model* http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model

[24] NVidia (2017), *CUDA Programming Model* http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#thread-hierarchy

[25] NVidia (2017), *Dynamic Parallelism* http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-dynamic-parallelism

[26] NVidia (2017), *Standard Library* http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#standard-library

[27] NVidia (2017), *CUDA Thrust* http://docs.nvidia.com/cuda/thrust/index.html

[28] NVidia (2017), *NVidia GPU Cloud* https://www.nvidia.com/en-us/gpu-cloud/