Towards Evaluating Many-Dimensional Physics Problems through Parallel Evaluation of Taylor Functions

Bachelor's Project Physics

July 2019

Author: M.J. van den Broek

Supervisor: Dr.ir. C.J.G. Onderwater

Second assessor: Dr. D. Roest

Abstract

Taylor series can be used in particle tracking experiments like the LHCb experiment, which investigates thousands of trajectories per second and requires an efficient evaluation algorithm. In this research a sequential and a parallel Taylor series evaluation algorithm are created. The algorithms can evaluate multiple multi-dimensional Taylor series at varying orders, using different input values. The performance of the algorithms at a varying number of evaluations and at varying orders is then investigated. The results show that an efficient parallel algorithm can be created. It can be further improved by reducing overhead.

Contents

1 Introduction

Mathematics is an indispensable tool in the arsenal of a physicist to describe phenomena and their relation to the environment. Mathematical functions can be used to determine the probable outcome of an experiment. If the functions hold true for every experiment testing it, then one can assume it accurately describes reality. These functions can then be used to logically deduce further theories. However, some of these functions can be very hard to solve. A tool to evaluate those functions is a series expansion. Differential equations are an example of this. They come up in physics involving harmonic oscillators or exponential decay for example and can be solved using a series expansion.

Most functions can be described by a sum of simple functions. Well known examples are the Fourier and Taylor series. A Fourier series describes a function as a sum of sines and cosines. A Taylor series, which will be the main focus of this research, describes a function as a sum of polynomials. Since functions are used in all fields of physics, Taylor expansions can be used in most of those fields as well. To evaluate the sum of a series expansion analytically, one can determine to what value the series converges.

In numerical evaluation only a finite number of terms of a series can be calculated. But a function expressed as a converging series can still be approximated with arbitrary precision. In a converging Taylor series, the higher order terms contribute less to the overall result. Depending on the accuracy needed, one must only solve a Taylor series up to a certain order. And a computer can solve this finite sum.

If a Taylor series is dependent on only one variable, *i.e.* a one-dimensional Taylor series, then a computer program executed on a modern central processing unit (CPU) can quickly evaluate this. If the series is dependent on several variables, *i.e.* multidimensional, then the number of terms increases significantly. A lot more terms need to be evaluated to achieve the same accuracy as in the one-dimensional case. A normal CPU based program calculates all these terms one at a time, stores the result of every term in memory and then adds them all up at the end. If this sequential evaluation needs to happen for thousands of terms it can take a lot of time. In experiments where time is valuable and series need to be evaluated very fast, a parallel algorithm can be more useful than a sequential algorithm.

A CPU core can only work sequentially, but a processor may have several cores that can work in parallel. The number of cores usually varies between 2 and 32 in current consumer CPUs. A graphics processing unit (GPU), however can have thousands of cores which can operate in parallel. As one can imagine, evaluating each term of a series in parallel could save a lot of time. This research intends to explore how a Taylor series evaluation can be sped up using parallelisation.

To determine the difference such a parallel approach has opposed to a sequential approach, two computer programs which can evaluate Taylor series are created. One program will run on a CPU and one on a GPU, i.e. one sequential and one parallel evaluation. The language in which the CPU algorithm is written is C++. The GPU algorithm is coded in CUDA C++, which is an extension on C++ created by NVIDIA that enables explicit commands for an NVIDIA GPU and its memory.

These algorithms will then be used to evaluate several Taylor series having different dimensions, which arise in different physics cases. One case involves the tracking of a particle through a magnetic field. This is an application, which is common in high-energy physics. In particular the LHCb experiment at CERN is investigated. The LHCb experiment analyses around a million collisions per second and can thus benefit from an efficient evaluation method.

To test the difference in efficiency between the parallel and sequential evaluation methods, the example cases are used to investigate how long both algorithms take to complete the evaluation. From this performance test, one can determine at what dimension and up to which order a parallel evaluation approach becomes faster.

The first thought may be that a parallel program is always faster than a sequential program. However, with GPU computation data is stored in both GPU and CPU memory and both memory types need to exchange data at some point. This creates overhead which might actually make it slower than a sequential CPU-based algorithm.

The goal of this research is to determine the relation in speed between sequential and parallel Taylor series evaluation algorithms. This information can then be used to determine if the increased efficiency of a parallel solution is sufficient for the intended application.

The research question is:

At what order and after how many evaluations of an n-dimensional Taylor series will a parallel algorithm become more efficient than a sequential algorithm?

2 Taylor series in physics

Taylor series come up frequently in physics. This section will elaborate on several examples from within physics that demonstrate how Taylor series are used. The examples will contain a series of increased dimensionality, which will show how a parallel evaluation algorithm becomes more relevant when a series is of a higher dimension. Precautions on accuracy versus computation speed are also discussed.

The definition of a single variable Taylor series expansion of an infinitely differentiable function is as follows [1, p. 353]:

$$f(x - x_0) = \sum_{n=0}^{\infty} \frac{1}{n!}(x - x_0)^n \frac{d^n f(x_0)}{dx^n}. \tag{2.1}$$

This expansion takes the form of a power series:

$$f(x - x_0) = \sum_{n=0}^{\infty} a_n (x - x_0)^n, \quad \text{where:} \quad a_n = \frac{1}{n!} \frac{d^n f(x_0)}{dx^n}. \tag{2.2}$$

The coefficients a_n define the function $f(x)$. They may be very difficult to evaluate depending on the function that is expanded. How the coefficients are derived will be discussed in the next subsection. However, the rest of this research will focus on evaluating Taylor series with known coefficients.

2.1 One-dimensional partial differential equation

A well-known case in physics is the harmonic oscillator. A simple example is solving the second order differential equation of an undamped harmonic oscillator. The motion of mass m oscillating on a spring with spring constant k is given by:

$$m \frac{d^2 x(t)}{dt^2} + kx(t) = 0. \tag{2.3}$$

Where x is the location at time t. The method for solving differential equations using power series is to take a solution of the form:

$$x(t) = \sum_{n=0}^{\infty} a_n t^n, \qquad \text{with its second derivative} \qquad \frac{d^2 x(t)}{dt^2} = \sum_{n=2}^{\infty} n(n-1) a_n t^{n-2}. \tag{2.4}$$

The assumption that this power series exists is valid since the requirement of a Taylor series is for a function to be infinitely differentiable and $x(t)$ satisfies that requirement. Substitution of the power series solution in equation 2.3 yields:

$$m\sum_{n=0}^{\infty}(n+2)(n+1)a_{n+2}t^n + k\sum_{n=0}^{\infty}a_n t^n = 0. \tag{2.5}$$

Both series contain t^n and have an equal summation range, therefore they can be combined to form the condition:

$$\sum_{n=0}^{\infty}[m(n+2)(n+1)a_{n+2} + ka_n]t^n = 0. \tag{2.6}$$

Equation 2.6 only has a solution if the part between brackets is zero for all $n \geq 0$, since $t^0 = 1$ for every t including $t = 0$. Solving for the part between brackets being equal to zero using a recursion relation, results in a solution for the even and uneven coefficients.

$$a_{2n} = (-1)^n\sqrt{\frac{k}{m}}^{2n}\frac{a_0}{(2n)!}, \tag{2.7}$$

$$a_{2n+1} = (-1)^n\sqrt{\frac{k}{m}}^{2n}\frac{a_1}{(2n+1)!}. \tag{2.8}$$

Putting these values back into equation 2.4 gives the final solution to the problem.

$$x(t) = a_0\sum_{n=0}^{\infty}(-1)^n\sqrt{\frac{k}{m}}^{2n}\frac{t^{2n}}{(2n)!} + a_1\sqrt{\frac{m}{k}}\sum_{n=0}^{\infty}(-1)^n\sqrt{\frac{k}{m}}^{2n+1}\frac{t^{2n+1}}{(2n+1)!}. \tag{2.9}$$

In this case the analytical solution is known and is given by:

$$x(t) = a_0 cos\left(\sqrt{\frac{k}{m}}t\right) + a_1\sqrt{\frac{m}{k}}sin\left(\sqrt{\frac{k}{m}}t\right). \tag{2.10}$$

Indeed, the Taylor expansion matches the analytical solution as both series in equation 2.9 represent cosine and sine respectively.

This method can be used to solve other differential equations as well. For example the second order differential Schrödinger equation of quantum mechanics or first order differential equations in exponential decay. The example above is a simple one and can be quickly evaluated on a computer. But in an experiment where numerous series need to be evaluated relating to many events happening every second, even simple one-dimensional problems might benefit from a parallel evaluating algorithm. Before an evaluation even begins however, one should make an important consideration.

The solution of equation 2.10 contains a cosine and a sine. To solve the cosine and sine series numerically requires the two series to be truncated. Calculating many terms results in a better approximation. However, this also increases the time it takes a computer to evaluate the series. To show how the quantity of terms determines the accuracy of the evaluation, the sine series is taken

as an example. As implied in equation 2.10, the sine expansion around $t = 0$ is given by:

$$sin(t) = \sum_{n=0}^{\infty} (-1)^n \frac{1}{(2n+1)!} t^{2n+1}. \tag{2.11}$$

The number of terms needed for an accurate evaluation of equation 2.11 depends on the value of t. This can be seen graphically in figure 1. If a system is only evaluated at very small values of t, then one or a couple of terms can suffice. At larger values of t more terms are needed. When numerically evaluating series, knowing the required domain of the variable in the series helps to determine how many terms are needed to achieve the desired accuracy. This is an important consideration as too few terms can result in a wrong evaluation and too many terms result in longer computation time.

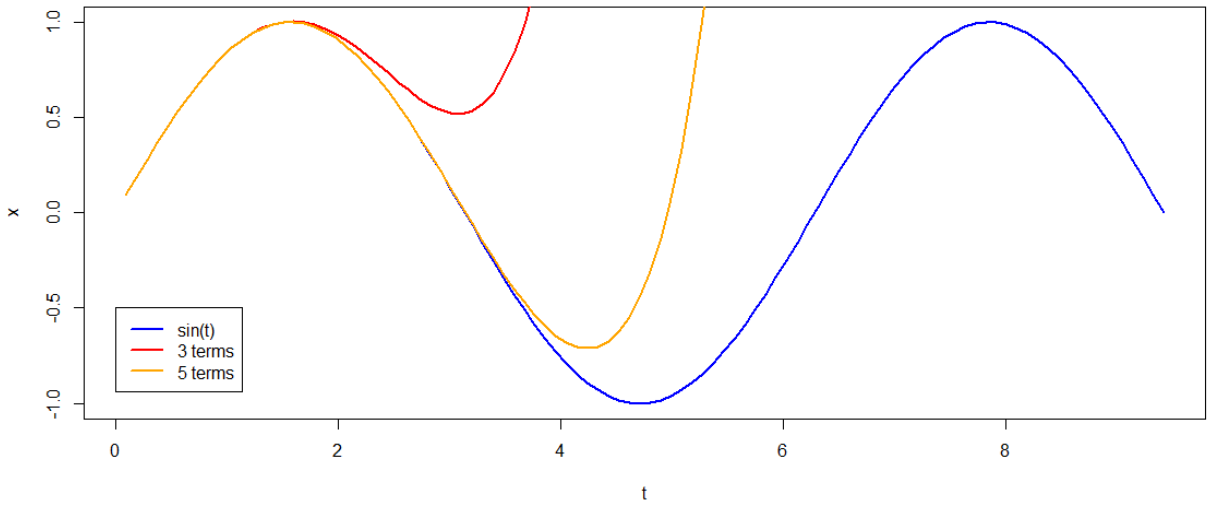

Figure 1: sin(t) approximated by three and five terms of the truncated Taylor series.

The expanded function is most accurate near the point at which it is expanded. In case a full period needs to be approximated by the sine series expansion around $t = 0$, the interval $[-\pi, \pi]$ would give the most accurate evaluation.

2.2 Multivariable Taylor series

Besides expanding a single variable function like in the example above, multi-variable functions can also be Taylor expanded. The expression of such an expansion is given below [1, p. 358].

$$f(x_1, ..., x_k) = \sum_{n=0}^{\infty} \frac{1}{n!} \left[\sum_{j=1}^{k} (x_j - b_j) \frac{\partial}{\partial x_j} \right]^n f(b_1, ..., b_k), \tag{2.12}$$

of which the coefficients b_j indicate the coordinates at which the function is evaluated. Equation 2.12 shows that when more variables are introduced, the number of terms increases due to the power over the second sum. The simplest multi-variable case has two variables, where at $n = 1$ the number of terms has already doubled, compared to the one-dimensional case. However, the extra terms which come with more variables also contain equal cross terms, which can be doubled instead of being calculated multiple times. For example, taking $b_1 = b_2 = 0$, $n = 2$ and $k = 2$ results in the part between square brackets in equation 2.12 being:

$$x_1^2 \frac{\partial^2}{\partial x_1^2} + x_1 x_2 \frac{\partial}{\partial x_1}\frac{\partial}{\partial x_2} + x_2 x_1 \frac{\partial}{\partial x_2}\frac{\partial}{\partial x_1} + x_2^2 \frac{\partial^2}{\partial x_2^2} = x_1^2 \frac{\partial^2}{\partial x_1^2} + 2x_1 x_2 \frac{\partial}{\partial x_1}\frac{\partial}{\partial x_2} + x_2^2 \frac{\partial^2}{\partial x_2^2}. \tag{2.13}$$

Equation 2.12 with only two variables is given below.

$$f(x,y) = \sum_{n=0}^{\infty} \frac{1}{n!} \left[(x - x_0)\frac{\partial}{\partial x} + (y - y_0)\frac{\partial}{\partial y} \right]^n f(x_0, y_0). \tag{2.14}$$

An example of a two dimensional system where a Taylor series can be used is the two-dimensional infinite square well of quantum mechanics. The method is very similar to the harmonic oscillator example from before. The equation that is to be solved is the Schrödinger equation, which is a partial differential equation in this case. To keep the example clear the constants in Schrödinger's equation have been combined in the constant k^2.

$$\left[\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right] \psi(x,y) + k^2 \psi(x,y) = 0. \tag{2.15}$$

The power series is used again as a general expansion of $\psi(x, y)$ around $(x, y) = (0, 0)$. It takes the form:

$$\psi(x,y) = \sum_{i=0}^{\infty} \sum_{j=0}^{\infty} a_{i,j} x^i y^j. \tag{2.16}$$

If the power series is evaluated up to order n, then the condition is $0 \leq i + j \leq n$. Using this condition, one can determine which terms belong to a certain order. Substitution of equation 2.16 into equation 2.15 and following similar steps as in the one-dimensional example yields:

$$\sum_{i=0}^{\infty} \sum_{j=0}^{\infty} \left[a_{i+2,j}(i+2)(i+1) + a_{i,j+2}(j+2)(j+1) + a_{i,j} \right] x^i y^j = 0. \tag{2.17}$$

Again the bracketed part needs to be equal to 0 for all combinations of i and j to solve the equation. This has become a lot harder than in the one-dimensional case, where a simple recurrence relation could be used. Solving the coefficients numerically by finding partial derivatives or by finding the recurrence relations both require a sequential approach. Recurrence relations rely on previously found equalities and evaluation of a higher order derivative requires the derivative one order lower. If a term in the series relies on values calculated in previous terms, they cannot be evaluated in parallel. Therefore a two dimensional numerical evaluation will focus on solving equation 2.16 where

the coefficients are predetermined values or independent functions. If predetermined values of the coefficients are known, then equation 2.16 can be filled in and the wave function can be directly evaluated. The wave function can also be derived in another analytical way explained in [2, p. 50]:

$$\psi(x, y) = A \cdot sin(c_x x) sin(c_y y). \tag{2.18}$$

A, c_x and c_y are independant functions, which are obtained within the derivation of equation 2.18. k^2 is missing from equation 2.18, but the constants contained in k^2 were included in c_x and c_y during the derivation. This function can again be evaluated using Taylor series as both sines can be expanded to form a power series like equation 2.11.

2.3 (LHCb) particle tracking

A case where multi-dimensional Taylor series can also be used is in tracking charged particles through a magnetic field. This kind of experiment is common in high energy physics, where the momentum of a particle can be determined by its trajectory. From the momentum and the trajectory the properties of a particle can be determined. In this section the LHCb experiment at CERN is considered. The experiment uses a particle accelerator, which collects 35 gigabytes of data from around one million collisions per second [3]. The algorithms currently used for particle tracking at LHCb use numerical integration methods, which are sequential.

However, a recent research used a Taylor series approach to particle tracking through an arbitrary magnetic field [4]. Initially this method predicted that the coefficients of the series could be determined beforehand and remain constant for successive evaluations. However, this turned out not to be true, requiring the coefficients to be determined for each trajectory. It is up for further research to find a method to calculate these coefficients as independent functions. Despite this, parallel evaluation of the Taylor expansion method for particle tracking could still be more time efficient than numerical integration. After the coefficients are known, the series can be evaluated in parallel making the method suitable for this research.

Following the approach presented in [4] for the LHCb experiment, the z direction is perpendicular to the detectors and the magnetic field is in the y direction as can be seen in figure 2. To determine the trajectory two expansions are needed describing the x coordinate as a function of z and the y coordinate as a function of z. The expansions take the following forms.

$$x(z) = \sum_{k,l,m} a_{klm}(z) \cdot x_i^{'k} \cdot y_i^{'l} \cdot x_3^m. \tag{2.19}$$

$$y(z) = \sum_{k,l,m} b_{klm}(z) \cdot x_i^{'k} \cdot y_i^{'l} \cdot x_3^m. \tag{2.20}$$

The prime denotes a derivative with respect to z and i says it is an initial value at $z = 0$. x_3 is the inverse of the circular radius of the trajectory. It has the property of containing the charge, momentum and magnetic field parameters in one variable. x_3 should not be equal to 0 as that would mean the particle would travel in a straight line, which is not very interesting for the experiment.

The z-dependence lies within the coefficients. The idea being that only the initial conditions need to be known to calculate the trajectory.

The above shortly describes the series which can be evaluated by a numerical algorithm where the coefficients a and b are known. This research focuses just on the part of evaluating equations 2.19 and 2.20. Further details on obtaining the coefficients and the exact derivation of x_3 can be found in the research paper [4].

The method used here for particle tracking has three variables, whereas the previous examples contain one and two variables. Evaluation of series with more variables is more intensive and the greater efficiency of a parallel approach should become more evident. The next chapter will discuss the considerations which are made to produce the parallel and sequential programs.

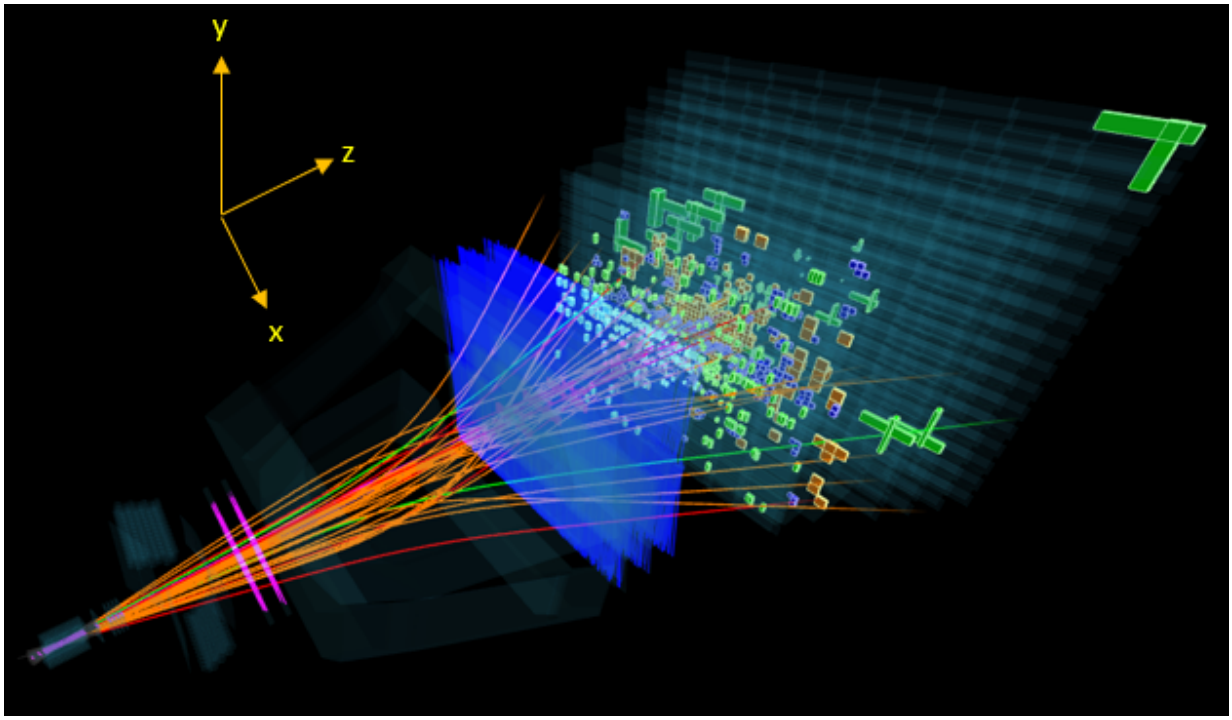

Figure 2: Collision event in the LHCb detector [5]

3 Concepts and requirements of the programs

The previous chapter discussed a few examples where Taylor series are used in physics. This chapter will focus on a computer algorithm to evaluate a Taylor series with one or several variables. The program is written in CUDA C/C++, which is an extension of C/C++, which enables utilisation of an NVIDIA graphics card present in the system. The syntax is similar to that of C and C++, except for commands which are related to the GPU. If further clarification about the CUDA C/C++ language is needed throughout this chapter, consider reading 'CUDA by Example' [6].

The program starts with the host function 'main', which prepares the input variables and delivers the output. From this main function other functions are called upon to perform calculations. These functions are assigned to run on the CPU or the GPU. If a function is executed by the CPU it is called a host function and if it is run on the GPU it is called a device function. The same applies to memory, where CPU memory is called host memory and GPU memory is called device memory. This chapter explains the sequential CPU algorithm and the parallel GPU algorithm divided into functions.

3.1 Sequential implementation

In the implementation of a sequential series evaluation algorithm a three-dimensional power series is considered. The series has the following form.

$$f(x,y,z) = \sum_{k,l,m}^{n} a_{klm} \cdot x^k \cdot y^l \cdot x^m, \tag{3.1}$$

The program first prepares input variables x, y, z, as well as the coefficients a_{klm}. The coefficient values are loaded from a file into an array. An output array, which will contain the evaluated Taylor series corresponding to each of the input variables, is initialised with all of its elements set to zero. The series corresponding to a set of input variables is evaluated by calculating every term one at a time and adding the result to the correct element in the output array. This is achieved using nested for-loops. The algorithm evaluating equation 3.1 is given below:

```
for (s = 0; s < SERIES_AMOUNT; s++){
   sums[s] = 0.0;
   tid = 0;
   for (h = 0; h <= ORDER; h++){
      for (m = 0; m <= h; m++){
         for (l = 0; l <= h - m; l++){
            sums[s] += a[tid] * pow(input[s].x, (h - m - l)) * pow(input[s].y, l) *
                pow(input[s].z, m);
            tid++;
         }
      }
   }
}
```

Where $sums[s]$ is the output array for each set of inputs s. l and m are the corresponding indices of equation 3.1. The index k is calculated as $h-l-m$ to satisfy the condition $k+l+m=h$, thus evaluating the sums sequentially for each successive order h.

The code shows a variable range of iterations of the inner loops. The outer loop runs from 0 to *order*, whereas the range of the inner loops depend on the value of the outer loops. The reason lies in the definition of the order. An evaluation up to a certain order in three dimensions requires $0 \leq k+l+m \leq order$. If all l, k, m went from 0 to *order*, a lot of higher order terms would be added and the total number of terms would be $l \cdot k \cdot m$. These higher order terms are less significant and are not needed to achieve the desired accuracy of the evaluation. When manually determining how many terms are calculated in a power series up to a certain order, a similarity to Pascal's triangle [7, p. 153] was found as visualised in figure 3.

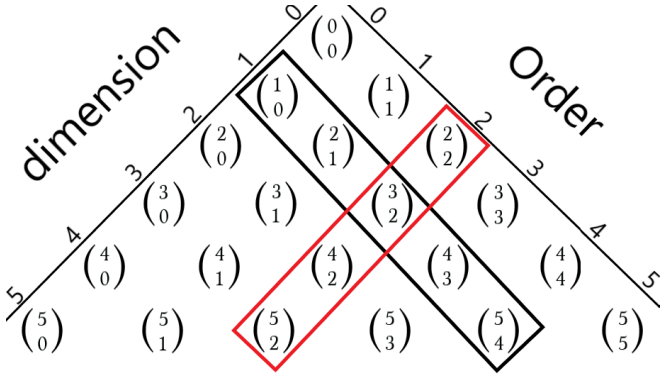

Figure 3: The number of terms that arise when evaluating a multidimensional power series up to a certain order is given by an element in Pascal's triangle.

The number of terms is then given by the binomial coefficient from figure 3 depending on the dimension and order of the series:

$$terms = \binom{order + dim}{order} = \frac{(order + dim)!}{dim! \cdot order!}. \tag{3.2}$$

For example, evaluating a three dimensional series up to $order = 4$ gives $\frac{(4+3)!}{4!3!} = 35$ terms. This value is used to determine how many coefficients need to be loaded into the array.

In testing the CPU algorithm it appeared that a self made power function was faster. This self made function used a for loop, which multiplies an input value by itself. The probable reason this is faster comes from the fact that the built in power function can handle fractions of powers. In this research only positive integer powers are needed and the self made function is only able to calculate positive integer powers. The power function in the code above is the built in power function, but

it serves as an illustration of what the algorithm looks like. The actual CPU algorithm will use the self made power function.

3.2 Parallel implementation

For the parallel implementation the three-dimensional power series of equation 3.1 is considered. The parallel equivalent of the sequential algorithm from above is less simple. The algorithm is divided into several device functions, to evaluate each term and to sum all evaluated terms. When a device function is called upon, a pre-specified number of the same function is started simultaneously, which are called Threads. To differentiate these Threads, each one has a specific ID. This is important since each thread uses a different set of values in its instruction. These IDs can be used as an index to address a location in an array for example. Before launching the device function one specifies how many threads with an ID in the x, y and z direction will be launched. The total number of thread IDs in all dimensions is limited to 1024 and the ID in each dimension may not exceed $(x, y, z) = (1024, 1024, 64)$. But threads are launched in blocks and these rules only apply to one block. Each thread inside a block has access to a memory cache contained within that block, this is called shared memory. That is why there is this separation between threads and blocks. Blocks also have an ID, but only in two-dimensions. These limits are much larger however $max(\text{blockIDx}, \text{blockIDy}) = (2^{16}, 2^{16})$. A visualisation of the concept of blocks and threads with indices in two dimensions is given in figure 4.

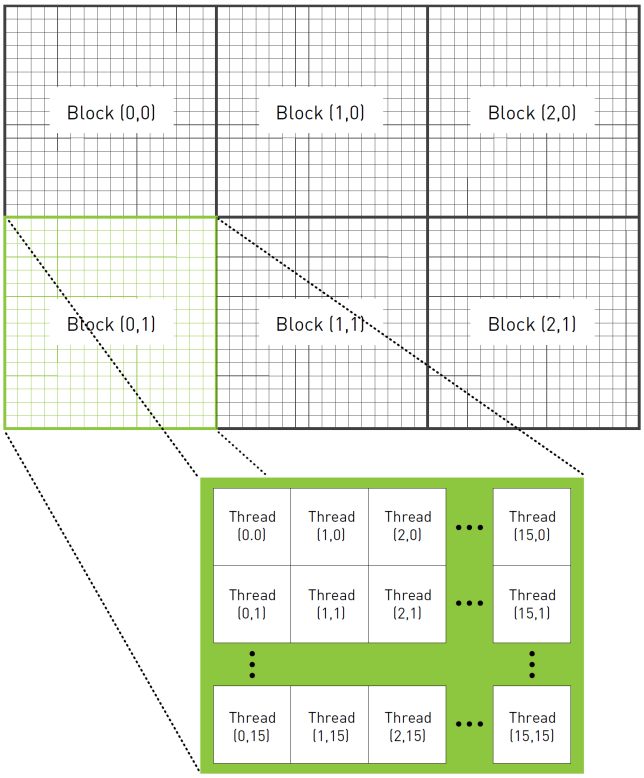

Figure 4: Grid of blocks and threads with two-dimensional indexing [8].

Multiplying block and thread IDs gives the range of indices available. A global ID in the x-direction can be calculated by threadIdx.x+blockIdx.x*blockDim.x, where blockDim.x is the number of threads in a block. The same goes for the y-direction. Global IDs in the z-direction only range from 0 to 63. Higher dimensions must be indexed using another way.

The index in the z direction can be obtained using the global ID of threads in the x direction. The number of threads in the x-direction must now be $(order + 1)^2$, since it is used for the x indices as well as the z indices. These indices are calculated within the device function as follows.

```
int globalID_x = threadIdx.x + blockIdx.x * blockDim.x;
int index_Y = threadIdx.y + blockIdx.y * blockDim.y;
int index_X = globalID_x % order;
int index_Z = globalID_x / order;
```

Where % is the modulus operator. Because integers are used a fraction is rounded down and the division gives the required integer result. If the maximum order is a power of two the computation-

ally heavy modulus operator can be replaced by a binary AND operator, which isolates the lower order bits from globalID_x. In that case also the division can be replaced by a less computationally intensive right shift operator, which shifts the binary digits to the right and thus isolates the higher order bits. Now that the indices are acquired they need to be used. The indexing required to address the correct values from the input arrays gives rise to some problems in the term evaluation function.

3.2.1 Term evaluation function

This function evaluates each term individually and stores the solution in an output array. This can be written as:

```
termSolution[termIndex] = a[termIndex] * pow(x,index_X) * pow(y,index_Y) * pow(z,index_Z);
```

Where index_X, index_Y and index_Z are the global indices. In the context of equation 3.1, index_X $= l$, index_Y $= k$ and index_Z $= m$. Recall that a self made power function is more efficient on the CPU than the built in function. The opposite seems to be true in the GPU case. Therefore the built in function is used in the parallel program.

When the function containing the code above is run it will start $l \cdot k \cdot m$ threads. However only the ones where $0 \leq l + k + m \leq order$ are needed. The x, y and z indices need to range from 0 to $order$. The number of threads launched is thus $(order + 1)^2$, where the $+1$ comes from counting the zeroth order. This could be overcome by using dynamic parallelism. This means that a parent device function is called and each thread calls another device function. Figure 5 provides a visual representation of how a thread-block launches additional functions from the GPU. For a 2-dimensional case, the parent device function has a number of threads equal to $order + 1$ and has thread IDs in only one dimension. The parent function then calls child device functions for each of the parent's threads. The number of threads of the child function is equal to the order minus the thread ID in the parent function. The thread ID of the parent cannot be passed to the child directly since this is a limitation of dynamic parallelism [9]. A global array containing values equal to its own index may be used to pass the Thread ID of the parent function to the child. The total number of threads can then be calculated using the binomial coefficient method from equation 3.2. The number of terms is needed to determine the size of the output array.

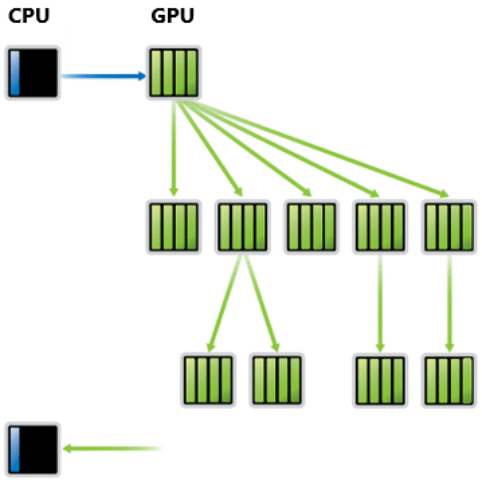

Figure 5: Visualisation of dynamic parallelism [10].

However, there is another problem. Every term needs to be stored somewhere in the output array. But every thread needs to have an index of the output array to store its value in a unique position. The index to the output 'termIndex', is also used in the coefficient array. In CUDA it is only possible to use one-dimensional array indexing. Therefore the global thread IDs in the x, y and z direction have to somehow be used to calculate the index, termIndex, of the one-dimensional output array. Take a two-dimensional series up to order three as an example. The table below shows the indices of the output array corresponding to the x and y indices.

		x index			
		0	1	2	3
y index	0	0	1	2	3
	1	4	5	6	n.a.
	2	7	8	n.a.	n.a.
	3	9	n.a.	n.a.	n.a.

Table 1: Indices of the termSolution array that correspond to the x and y thread indices

Calculating termIndex can be done with the use of a separate index array. This index array can be calculated as follows:

```
indexArray[0]=0;
for(i=1;i<order;i++){
    indexArray[i]=indexArray[i-1]+order+2-i;
}
```

What is calculated is simply the left collumn of table 1. The way termIndex is determined is by

adding index_X and indexArray[index_Y]. The loop to calculate indexArray requires a previously determined value to calculate the next value. Therefore it cannot be implemented in parallel. But it can be calculated beforehand and loaded into an array when the program is needed, since this array remains the same when the dimension and order of the series remain the same. In a 3D case however, indexArray becomes a 2D array, which is also calculated using a recursion loop. It holds the left collumn as in the 2D case, but it does so for every index_Z. The indexArray has the same form as seen in table 1. Another indexArray is thus needed to get the correct index in the indexArray. For higher dimensions the number of indexArrays needed is equal to the dimension minus one.

Since the above solution requires many sequential calculations and requires a lot of memory space in higher dimensions due to the size and number of indexArrays it is not a good solution. Another way to calculate termIndex is by using the Cantor tuple function [11]. It contains a flooring function $\lfloor x \rfloor$, which rounds x to the lowest integer. It requires a different ordering than in table 1, namely:

		x index			
		0	1	2	3
y index	0	0	1	3	6
	1	2	4	7	n.a.
	2	5	8	n.a.	n.a.
	3	9	n.a.	n.a.	n.a.

Table 2: Reordered indices of the termSolution array that correspond to the x and y thread indices

In this arrangement termIndex increases sequentially along each successive diagonal, which corresponds to successive orders, because $x + y$ remains constant along each diagonal. The Cantor tuple function $\pi(x, y)$ can be used to calculate termIndex in two dimensions and states [11]:

$$\text{termIndex} = \pi(x, y) = \left\lfloor \frac{(x + y + 1)(x + y)}{2} + y \right\rfloor. \tag{3.3}$$

Where x and y are the indices. For three dimensions a similar function was devised:

$$\text{termIndex} = \left\lfloor \frac{(x + y + z + 2)(x + y + z + 1)(x + y + z)}{6} + \frac{(y + z + 1)(y + z)}{2} + z \right\rfloor. \tag{3.4}$$

With this equation termIndex increases sequentially along the diagonal plane, where $x + y + z$ is constant. For higher dimensions a general function was devised, which can be found in the appendix. However, the three-dimensional function is significantly more complex than the two-dimensional function. Using such an elaborate function in each thread will reduce performance. Another solution to the indexing problem can be found using a different approach.

Instead of calculating termIndex using the x, y, z indices, one could try finding the indices x, y, z

from termIndex. In this approach the ordering of termIndex from table 2 is used, where each order corresponds to a diagonal. Using a nested for-loop the indices can be determined from the value of termIndex. For the three-dimensional series the code is as follows:

```
int termIndex = 0;
for (h = 0; h <= ORDER; n++)
{
   for (m = 0; m <= n; m++)
   {
      for (l = 0; l <= h - m; l++)
      {
         factors[termIndex].k = h - l - m;
         factors[termIndex].l = l;
         factors[termIndex].m = m;
         termIndex++;
      }
   }
}
```

The array 'factors' is a struct (data structure) containing the indices k, l, m from equation 3.1 and the coefficients a_{klm}. The coefficients are loaded in seperately. When this loop completes the factors array contains the correct indices and coefficients belonging to termIndex. Notice that the loops are the same as in the sequential algorithm without its outer loop. The justification of this sequential solution to the indexing problem is that the code needs to run only once. After the values have been loaded in, the factors array can be used for the series evaluation of all sets of input variables x, y, z. Therefore the factors array can be stored in constant device memory.

Constant device memory has the ability to broadcast a single memory call to multiple threads. This decreases the memory latency. However, it has a limited capacity of 64kB [12, p. 250]. A three-dimensional series can go up to the 22^{nd} order. If a higher order is needed, the factors array can also be stored in global device memory. Although global device memory is much larger than constant memory, it has a performance cost.

The dimensional variables are also stored in a struct, since the initial values are dependent on each other. In the LHCb experiment for example, a collision produces a set of initial conditions for each trajectory. With this struct of the input variables only one parameter needs to be passed instead of three.

The dimensional variables, coefficients and proper indices have been prepared and can be loaded into the term evaluation function. The function is not started from the CPU, but dynamic parallelism is used. The program should evaluate power series expansions with multiple sets of dimensional variables. Therefore a parent device function is started first (see appendix). The number of threads of the parent function depend on the number of series that need to be evaluated. From the parent function the term evaluation function is called. Each parent thread passes its global thread ID to the term evaluation function. This is used to index the correct dimensional variables. The term evaluation function requires thread IDs in only one dimension and is given below.

```
int tid = threadIdx.x + blockDim.x * blockIdx.x;
terms[tid + gridDim.x * blockDim.x * ptid] = factors[tid].a *
    pow(input[ptid].x, factors[tid].fac_x) *
    pow(input[ptid].y, factors[tid].fac_y) *
    pow(input[ptid].z, factors[tid].fac_z);
```

Where tid is the global thread ID and has replaced termIndex. Terms is the output array containing every term. GridDim.x is the number of blocks called by the parent thread. And ptid is the global thread ID of the parent function.

The index of the output array terms adds an offset to tid. This is the offset to the next series evaluation using different dimensional variables. The number of terms in a single series evaluation can be smaller than this offset. The output array will therefore have some unused padding elements. It is not always possible to adjust the number of threads and blocks to exactly fit the number of terms. Therefore the size of the output array is determined by the number of threads and blocks per term evaluation function call, and the number of parent function calls. All elements are then set to zero to ensure the term addition function of the next subsection produces correct solutions.

The sizes of the factors and input arrays is also determined by the number of threads and blocks of the term evaluation call. Their padding elements are set to zero to make sure the padding elements in the output array remain zero.

3.2.2 Term addition function

The parallel term addition algorithm has already been covered by others and is mentioned in GPU Gems 3[13, ch. 39], where it is called a parallel prefix sum algorithm. This section will explain the implementation in this research.

The function takes an array as input. Each thread stores an element in the shared memory of its block. Remember that this memory is specific to each block and cannot be accessed outside of that block. The number of elements added is equal to the number of threads. The correct elements are obtained by giving each thread a global ID as explained at the start of section 3.2. Only thread IDs in the x-dimension are needed since the function sums a one dimensional array. After this first addition the threads within each block are synchronised to make sure all additions are complete before moving on to the second addition. For this second addition, each thread adds two elements from shared memory and stores the result by overwriting the shared memory element that has the lowest index. After addition all threads are synchronised again. These synchronisations must be done to avoid a thread being one addition ahead and an already added element is added twice. The process continues until all elements in the shared memory are added. Each block now has one output value which is stored in an output array. The index of the output array is determined by the ID of the block. Figure 6 illustrates the process.

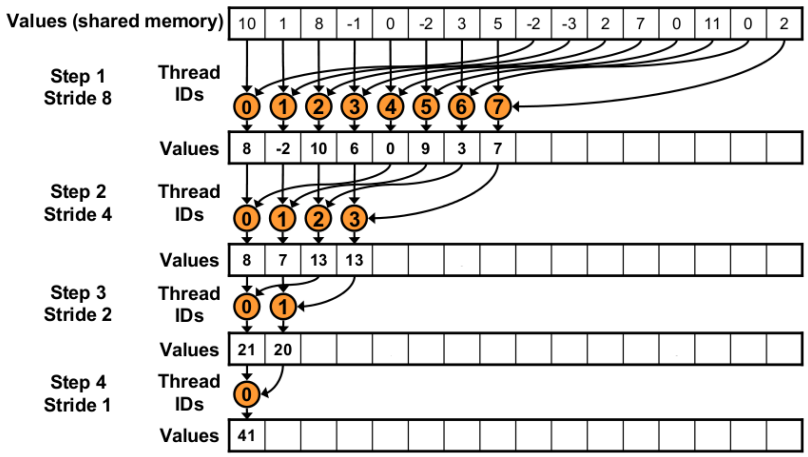

Figure 6: The process of adding elements in an array [14].

In the figure an even number of terms is added. In case the number of terms is uneven, one element must be left untouched. Figure 6 shows that the memory elements on the left are updated. However, the blank elements in the figure are not empty and contain terms that were added in a previous iteration. If an uneven number of terms is treated as an even number of terms, then one element would be added twice or one element would be left out of the addition. Therefore an if-statement is used to check for an even or uneven number of terms per addition.

If-statements result in branching which cause a parallel program to perform each branch sequentially. In case of an uneven number of terms in the addition, one thread will not pass the if-statement. That thread will do nothing and the odd element remains in shared memory without an addition being made to it. This branching should not be a performance issue since one of the branches is to do nothing and will therefore not require additional operations when performed sequentially.

As can be seen in the figure a lot of memory calls are being made. Shared memory is used because it has a latency which is around 100 times lower than that of the global GPU memory [15]. The cost in time from input/output operations is thus greatly reduced. The maximum shared memory size is 48kB and higher depending on the GPU [12, p. 250]. The maximum array size loaded into shared memory in the term addition function is the number of threads times the size of a double precision variable or double. The maximum number of threads per block is 1024 and a double consist of 8 bytes. Thus the maximum shared memory space is 8kB. This is well within the acceptable size.

If the function call needs more than one block, the term addition function must be called again. Now with the output array of the previous term addition function call as its input. The function needs to be called repeatedly until the number of additions is equal or lower than the maximum number of threads inside one block. This because each block only adds the terms in its own shared memory.

To sum all terms of one series, the term addition function is called from the same parent thread that called the term evaluation function. The same number of threads and blocks as in the term evaluation function are called. Recall that the padding elements ensured that the number of threads times the number of blocks from one function call will hold the terms from only one series. The padding elements are zero and will not contribute in the term addition function.

The input array of the term addition function holds the terms of all series. The term addition function distinguishes terms from different series evaluations by introducing an offset when loading elements into shared memory. This offset is dependant on the thread ID of the parent function and the number of threads and blocks with which the term evaluation function was called. Storing the output in a unique location in the output array depends on an offset based on the parent thread ID and the number of blocks.

The code of the term addition function can be found in the appendix.

3.3 Precautions

CUDA C/C++ was created by NVIDIA and is only supported by graphics cards from that company. Their graphics cards have a specification called compute capability. As the name suggests, if this specification has a higher number it supports more functions. Current graphics cards have a compute capability high enough to support everything discussed above. However, one should be aware of this specification on older hardware. For example dynamic parallelism requires compute capability 3.5. This feature requires the highest compute capability used in this research.

Continuing with double precision variables, they can hold a value with a maximum of 15 decimal places. Terms contributing to lower decimal places contribute nothing. In case that kind of precision is not necessary one can adjust the program to perform its calculations with single precision variables, or floats, which occupy 4 bytes, thus reducing the required memory by two. Floats can hold values with up to 6 decimal places. The range of the exponent in a double is $[2.22507E-308; 1.79769E+308]$ and the range in a float is $[1.17549E-38; 3.40282E+38]$, which is the same for negative values. This can be simply checked by requesting the variable properties from the compiler.

Besides these GPU and language precautions, NVIDIA is slowly removing support for 32-bit systems. If a 32-bit system is used, a compatible version of the CUDA software as well as a compatible graphics card must be used.

4 Program results on series of different complexities

The previous chapter discussed a sequential and parallel approach to evaluate a power series numerically on a CPU or GPU. To answer the research question, both algorithms are tested at varying orders and dimensions and for a different number of evaluations. The results of the tests are then given and discussed. The examples from chapter 2 resemble the test cases.

4.1 Test conditions

The performance of the algorithms can change with different hardware. The system specifications used in this research are the following. The CPU is an Intel CORE i5-9600k with a clock-speed of 4.7 GHz. The GPU is an NVIDIA GeForce 970 with a clock-speed of 1.22 GHz. The total device memory is 4GB and the total host memory is 16GB.

The algorithms are investigated using 50, 500, 5,000 and 10,000 evaluations per execution. The set of input variables in each of those evaluations is arbitrary but unique. This means that every evaluation requires a unique set of input variables to be placed in memory, instead of one set being used multiple times. This will ensure the same memory latency, *i.e.* overhead, is generated as in a normal experimental scenario.

The input variables and the coefficients are supplied to the algorithms in the same manner. Therefore, the latency caused by initialising these variables is the same for both programs. Differences in run-time are dependant on calculations and memory operations beside initialising input variables and coefficients.

Every execution of the algorithms is performed multiple times. The average run-time is used as a result and the largest deviation from the average is taken as the error. The error arises from secondary computer activity. During execution all possible side processes running on the system are closed. But the system must run the operating system and other necessary programs. These essential processes have an influence in the run-time of the tests. The error is greater in the tests of the parallel algorithm, as the GPU is processing display output whilst executing the parallel program. The parallel algorithm run-time is influenced by display processes and essential system processes.

The algorithms were tested for correctness by performing a low order evaluation and comparing the output with the correct result. This was done at low order, since then it is possible to evaluate the correct result manually.

4.2 Results

4.2.1 One-dimensional results

The example from chapter two requires solving equation 2.10, which contains a sine and a cosine. For the one-dimensional test the sine series is evaluated. The coefficients for the sine function are calculated beforehand and can be found in equation 2.11. Only 85 coefficients could be calculated within the precision of a double. After 85 coefficients the values become too small to be contained in a double and thus become zero. These terms will not contribute to the outcome of the evaluation, but they will take part in the calculations. This way the performance at orders higher than 84 can be measured. The 84^{th} order includes the 0^{th} order and thus contains 85 coefficients. The series is evaluated at orders ranging between 10 and 1000.

The test of the one-dimensional sequential algorithm is shown in figure 7, which is a logarithmic plot.

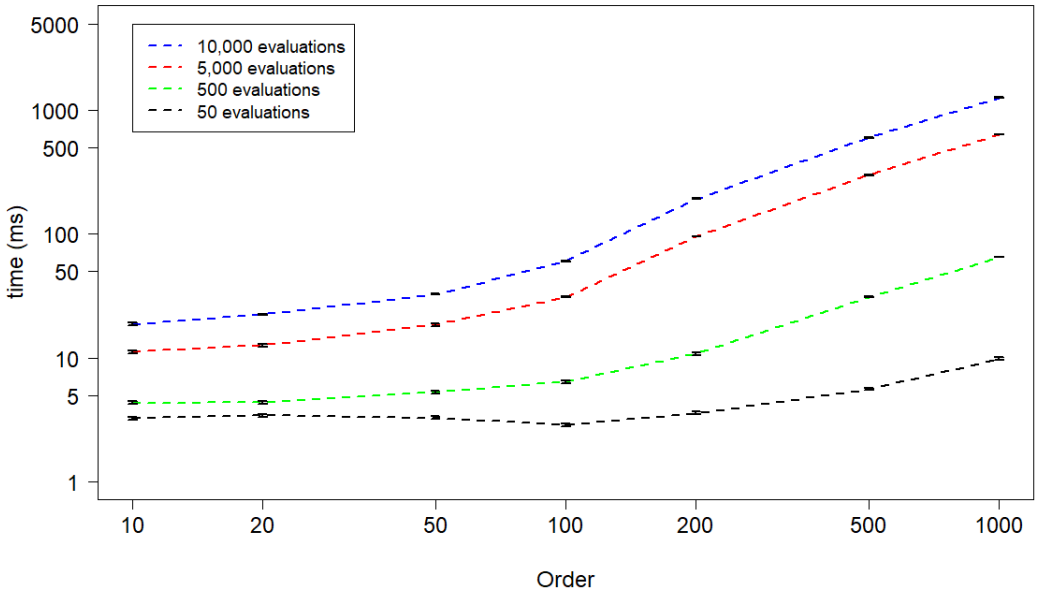

Figure 7: Run-time against evaluation order of the one-dimensional sequential algorithm at varying numbers of evaluations

Figure 7 shows a shallow slope up to around the 50^{th} order of the executions with 5,000 and 10,000 evaluations. This is caused by overhead. The time required to arrange and transfer the variables is more significant than the time required to perform the calculations. After around the 50^{th} order

the slopes start to increase. Around that point the time required for computations becomes more significant compared to the overhead. The largest error in run-time due to secondary computer processes is 3%.

The results of the one-dimensional parallel algorithm are shown in figure 8.

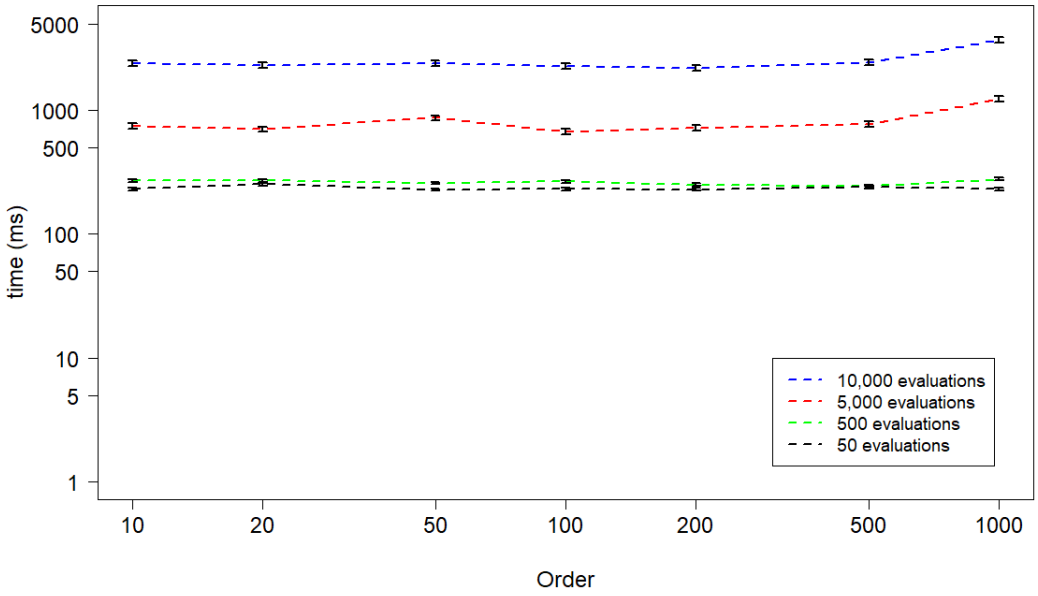

Figure 8: Run-time against evaluation order of the one-dimensional parallel algorithm at varying numbers of evaluations

As can be seen in figure 8, the run-time remains almost constant as the order increases. At a higher order, the number of computation is also higher. Performing computations is therefore not a significant factor in the total run-time. However, the parallel program is slower than the sequential algorithm. It appears the significant factor in the total run-time is overhead. Since the run-time increases significantly at a higher number of evaluations and not at a higher order, a part of the overhead comes from memory transfers of the input variables.

4.2.2 Two-dimensional results

For a two-dimensional series the case of chapter two was a particle in a square well. Two sines are multiplied, which can be expanded by a single two-dimensional series. The series was evaluated up to order 80 instead of 1000, which was the order in the one-dimensional test. The reason is the limited size of constant device memory in which the coefficients and factors are stored.

Besides the limit of constant memory, a hardware limit was also reached. When increasing the number of evaluations beyond what was used in the tests, a memory error occured and the display would turn black for a short moment. At the 80^{th} order, the limit in the number of blocks and threads is still in the acceptable range. One evaluation at the 80^{th} order contains 3,321 terms, which is calculated using equation 3.2. This requires 7 blocks with 512 threads of the term addition and summation functions. At 10,000 evaluations, 20 blocks of 512 threads are required for the parent function.

The performance of both algorithms in the two-dimensional case is similar to the one-dimensional case. The results of the two-dimensional test are shown in figures 9 and 10 in the appendix.

4.2.3 Three-dimensional results

The particle tracking method described in chapter two uses a three-dimensional series. The evaluations of this series were performed with an arbitrary set of coefficients. The orders up to which the series were evaluated ranged from 5 to 20. The same arguments for the smaller order as in the two-dimensional test apply.

The results are similar to the one and two-dimensional cases. Computations of increasing order are not a significant cause in latency in the case of the parallel program, where overhead remains the primary cause of latency. The results of the three-dimensional test are shown in figures 11 and 12, which are found in the appendix.

4.3 Discussion

The tests in each dimension were performed up to the maximum order and maximum number of evaluations possible. This limit caused the maximum number of terms in the tests of different dimensionality to be approximately the same. Therefore, similarities are found in the result plots of different dimensions.

The errors produced during the tests of the parallel program could be further reduced by using a GPU which is specifically used for series evaluation and does not perform any secondary activity. In this case the hardware limitation is reduced as more memory is available in testing the program.

Since the limitation in the number of calculated terms is probably limited by the hardware, a GPU containing more memory could reduce this limitation. The run-time of the sequential algorithm increased at an increasing order, whereas the parallel algorithm showed no significant differences in run-time at an increasing order. If the aforementioned limitation is reduced a test could observe the parallel algorithm being faster than the sequential algorithm.

The results showed that the most significant cause of latency was overhead. Further improvement of the parallel program should therefore be focused on reducing this overhead. A solution could be to use an integrated GPU. This means that the GPU and CPU share the same host memory.

5 Conclusion

Particle tracking at the LHCb experiment requires fast evaluation algorithms, since thousands of trajectories need to be evaluated every second. A Taylor series method derived in a previous research [4], provided the possibility for a parallel algorithm. To ensure a parallel approach in solving Taylor series can be made efficiently, this research aimed at comparing the evaluation speed of a parallel to a sequential algorithm.

To make a parallel evaluation program, an efficient indexing procedure was required. Several indexing procedures were considered. First the indices were calculated during the parallel evaluation. This was achieved using the Cantor tuple function. This function was then extended to enable index calculations at dimensions higher than two. However, at these higher dimensions the index computation significantly increased in complexity. A different approach was to calculate the indices sequentially before the parallel evaluation and pass them as parameters. This was justified, because the index determination needs to be performed once. It can then be used for a large number of evaluations. The time required to evaluate the indices is equal to the time of the sequential series evaluation program to evaluate one series.

To test the efficiency of the sequential and parallel algorithm tests were performed using one, two and three dimensional series evaluations. The sequential program required more time to complete the tests as the order and the number of evaluations increased. The significance of overhead on latency decreased as the series were evaluated to a higher order.

The results of the parallel program showed that the program is efficient at performing a high number of computations. The run-time was not significantly affected by the increased number of computations arising from evaluations of increasing order. The significant factor in latency was overhead. Since the run-time was not significantly affected by the order, the difference in run-time between tests using a different number of computation is caused by overhead in input variables. A higher number of evaluations requires a higher number of input variables which need to be transferred through memory.

A limit in the total number of term evaluations that could be performed appeared to originate from a hardware limitation. Therefore, the maximum order decreases with tests using increasing dimensionality.

To conclude, the research question is answered based on the obtained results. The question was: *At what order and after how many evaluations of an n-dimensional Taylor series will a parallel algorithm become more efficient than a sequential algorithm?* The parallel program was not more efficient than the sequential program within the limit in number of computations possible during the tests. At the highest number of evaluations and at the highest order tested, both algorithms performed similarly. The parallel algorithm could perform an increasing number of computations without significantly increasing latency. The reason the parallel algorithm was less efficient was due to overhead.

Appendix

Multi-dimensional Cantor tuple function

A recursive function can be devised, derived from the Cantor tuple function, to determine the termIndex at dimensions higher than two.

$$f^{(n+1)}(x_0, ..., x_n) = \left\lfloor \frac{\prod_{j=0}^{n}[(\sum_{i=0}^{n} x_i) + j]}{(n+1)!} + f^{(n)}(x_1, ..., x_n) \right\rfloor, \tag{5.1}$$

where f corresponds to termIndex and x_i are the dimensional indices. Using starting value of $f^{(0)} = 0$ (termIndex for zero-dimensional case is 0), the one-dimensional case becomes $f^{(1)}(x) = x$ (sequential termIndex) and the two-dimensional case $f^{(2)}(x, y)$ equals the Cantor tuple function stated in equation 3.3:

$$\text{termIndex} = f^{(2)}(x, y) = \left\lfloor \frac{(x + y + 1)(x + y)}{2} + y \right\rfloor. \tag{5.2}$$

Code

Parent function

The parent function from which the term evaluation and term addition functions are called.

```
__global__ void parentEvaluation(DIM_VAR *input, double *terms, int *BlockCount, int
    *ptid, double *sums, double *sumStore)
{
   int tid = threadIdx.x + blockDim.x * blockIdx.x;

   //call term evaluation function
   termEvaluation << <BlockCount[0], TpB >> > (input, terms, ptid[tid]);

   //call term addition function
   sumTerms << <BlockCount[0], TpB >> > (terms, sumStore, ptid[tid]);
   sumTerms << <1, d_BlockCount[0] >> > (sumStore, sums, ptid[tid]);
}
```

DIM_VAR is the struct name of the input array containing the dimensional variables. Terms is the output array. BlockCount determines how many blocks of the term evaluation and sum addition functions need to be called. Ptid contains the thread ID of the parent function. SumStore contains the temporary sum of the term addition function when multiple blocks are called. Sums is the

output array containing the final solution to the evaluated power series. Tid is the global thread ID of the function. TpB stands for threads per block and is defined as a global constant.

Term addition function

```
__global__ void sumTerms(double* inf, double* outf, int ptid)
{
   //shared memory allocation
   __shared__ double temp_sum[TpB * sizeof(double)];

   //perform first addition of elements and load the result into shared memory
   int j = blockIdx.x * blockDim.x + threadIdx.x;

    //load elements from input array into shared memory
   temp_sum[threadIdx.x] = inf[j + (gridDim.x * blockDim.x * ptid)];
   __syncthreads();

    //add elements from shared memory
   int n = blockDim.x;
   int n2;
   while (n > 1)
   {
      n2 = n;
      n = (n + 1) >> 1;
      if (threadIdx.x < (n2 >> 1))
      {
         temp_sum[threadIdx.x] += temp_sum[threadIdx.x + n];
      }
      __syncthreads();
   }

   //The sum of the elements per block is written to the output array by thread 0
   if (threadIdx.x == 0)
   {
      outf[blockIdx.x + gridDim.x * ptid] = temp_sum[0];
   }
}
```

Inf is the input array, outf is the output array and ptid is the thread ID of the parent function.

Results

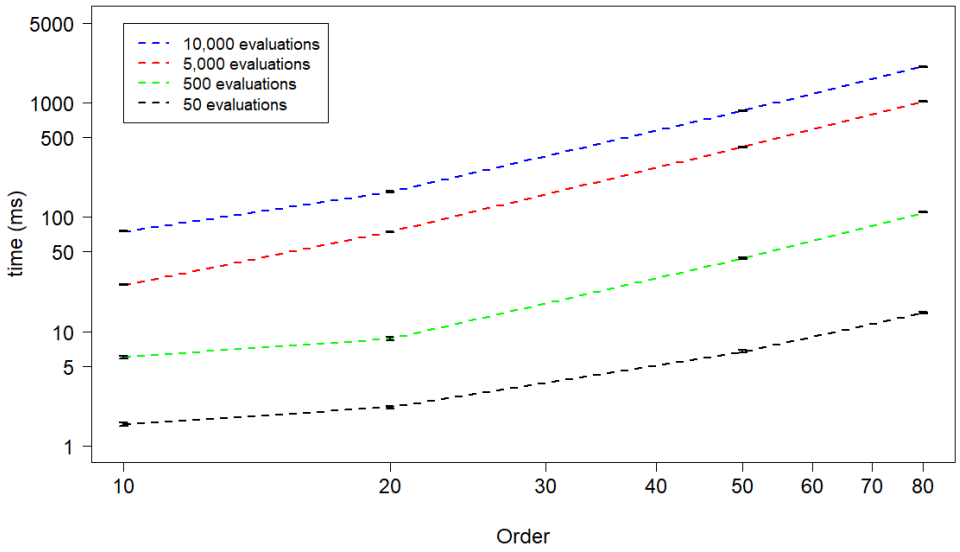

Figure 9: Run-time against evaluation order of the two-dimensional sequential algorithm at varying numbers of evaluations

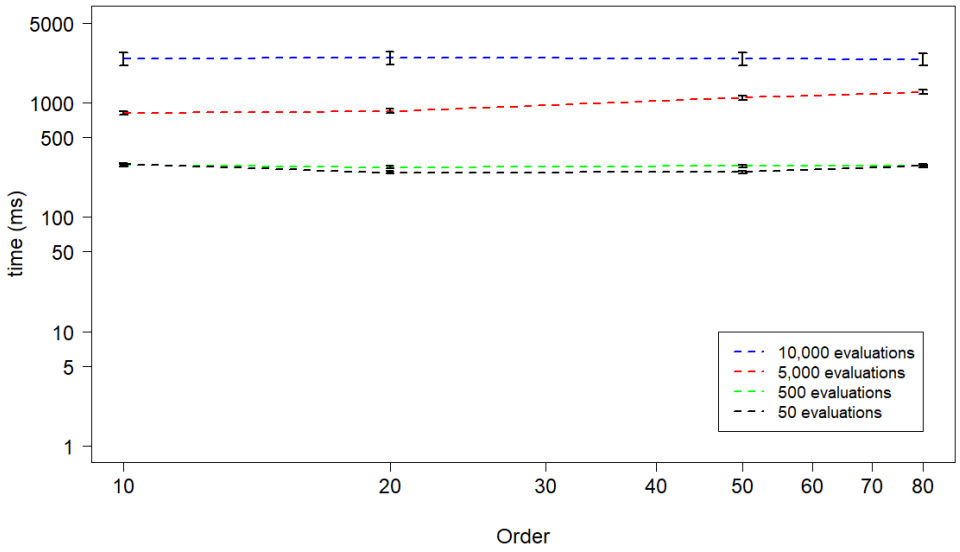

Figure 10: Run-time against evaluation order of the two-dimensional parallel algorithm at varying numbers of evaluations

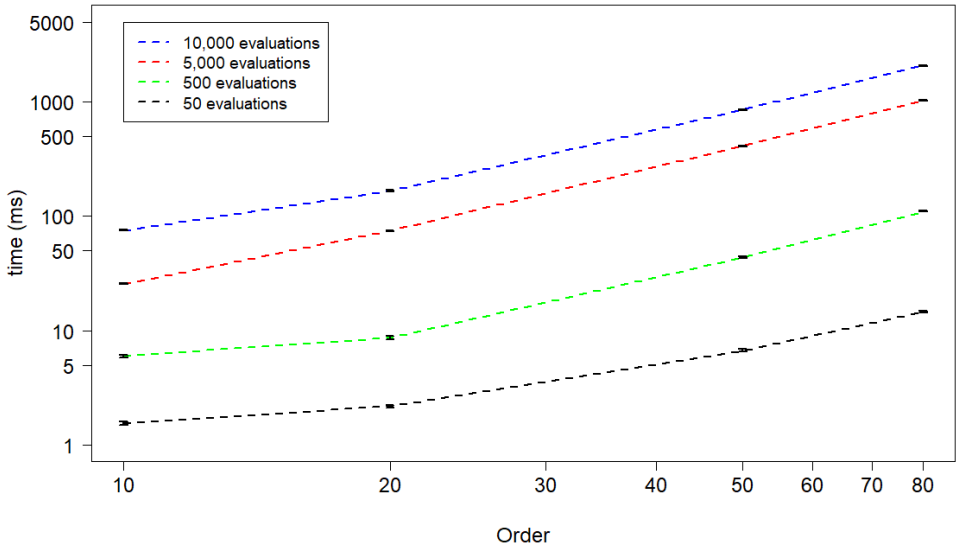

Figure 11: Run-time against evaluation order of the three-dimensional sequential algorithm at varying numbers of evaluations

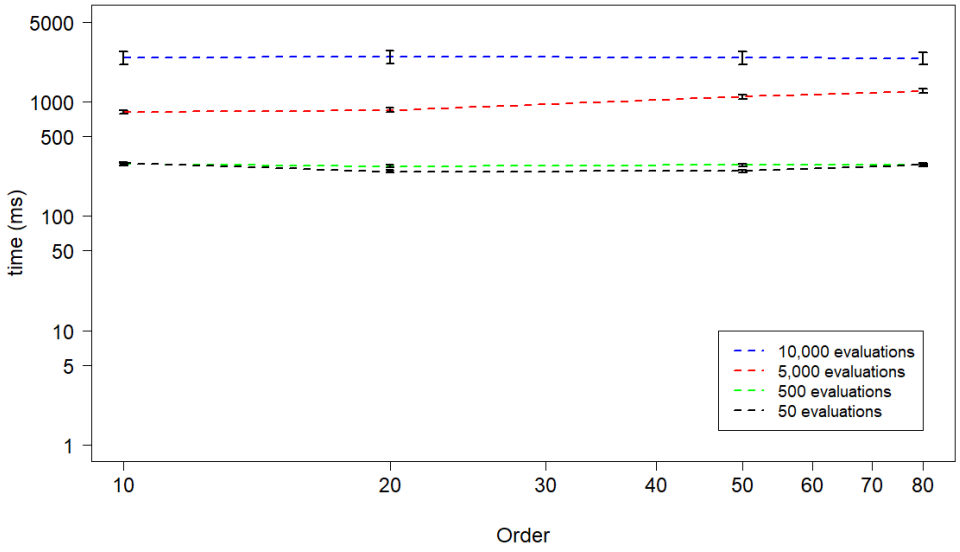

Figure 12: Run-time against evaluation order of the three-dimensional parallel algorithm at varying numbers of evaluations

References

[1] George B. Arfken Hans J. Weber. *Mathematical Methods for Physicists*. Elsevier Academic Press, 2005. ISBN: 0120885840.

[2] Y.B. Band Y. Avishai. *Quantum Mechanics with Applications to Nanotechnology and Information Science*. Elsevier Academic Press, 2013. ISBN: 9780444537867.

[3] LHCb. *LHCb data collection*. URL: https://lhcb-public.web.cern.ch/lhcb-public/en/Data%20Collection/Triggers-en.html. (accessed: 29.05.2019).

[4] Maurice Dekker. "Using Taylor series for fast and precise charged particle tracking in the LHCb magnet". In: (2018). Unpublished bachelor thesis, University of Groningen.

[5] Stefania Pandolfi. *CUDA Programming: thread, block, grid*. URL: https://phys.org/news/2017-03-lhcb-exceptionally-large-group-particles.html. (accessed: 02.07.19).

[6] Jason Sanders Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2011. ISBN: 978013138768.

[7] Pure and Applied Mathematics. *Fibonacci and Lucas Numbers with Applications*. Vol. 1. John Wiley Sons, Inc, 2001. ISBN: 047139969.

[8] *CUDA Programming: thread, block, grid*. URL: https://www.researchgate.net/figure/Dynamic-Parallelism-technology-22_fig3_315830821. (accessed: 30.06.19).

[9] Andy Adinets. *CUDA Dynamic Parallelism API and Principles*. URL: https://devblogs.nvidia.com/cuda-dynamic-parallelism-api-principles/. (accessed: 18.06.2019).

[10] Mohammed A. Shehab. *A Hybrid CPU-GPU Implementation to Accelerate Multiple Pairwise Protein Sequence Alignment*. 2017. URL: https://www.researchgate.net/figure/Dynamic-Parallelism-technology-22_fig3_315830821.

[11] John E. Hopcroft Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979. ISBN: 020102988.

[12] NVIDIA. "CUDA C Programming Guide". In: (2019).

[13] Huber Nguyen. *GPU Gems 3*. Addison-Wesley, 2008. ISBN: 9780321515261.

[14] *Calculate the sum of values in an array using renderscript*. URL: https://www.researchgate.net/figure/Dynamic-Parallelism-technology-22_fig3_315830821. (accessed: 30.06.19).

[15] Mark Harris. *Using Shared Memory in CUDA C/C++*. 2013. URL: https://devblogs.nvidia.com/using-shared-memory-cuda-cc/. (accessed: 17.06.2019).