Toward an Automatic GPU Parallelization of the Crank-Nicolson Method for Solving Time-Dependent Partial Differential Equations in Financial Applications

Jacob Jepsen

Department of Computer Science, University of Copenhagen (DIKU)

jepsen@diku.dk

Abstract

We present our experience in parallelizing a financial program that solves partial differential equations used in computational finance. The goal is to determine which optimizations and transformations are the most profitable to perform and if they can be done automatically by a compiler.

We ported the program from C to OpenCL (by hand) for GPU execution. Since the original program does not exhibit perfect-loop nests, a naive translation would parallelize only one of the loops, which would significantly under-utilize the GPU hardware. Instead, we use loop distribution and loop interchange to form parallel, perfect-loop nests, thereby increasing the degree of parallelism, and we recount the dependency analysis theory that guarantees the safety of these transformations.

Next, we investigate lower-level optimizations related to locality of reference, that were also performed by hand. After benchmarking both the C and OpenCL code, we identify that memory coalescing is the most profitable such optimization, and accounts for almost all of the speedup.

We conclude that performing the transformations is relatively straightforward, and requires little analysis for many programs. Thus a compiler implementation performing these transformations should be possible.

1 Introduction

In this report we examine a real-world implementation of a financial program written in low-level C, and we examine the code transformations that optimizes the execution on GPU hardware. We present the transformations that we have found to be the most profitable and assess their contribution to speedup. The program solves a set of partial differential equations of a financial asset pricing model using the Crank-Nicolson approximation method [6].

As is often the case with financial applications, our target program is a big-compute, big-data problem. Parallelization on GPU and related optimizations improve the total running time significantly, which may allow for faster real-time decisions to be taken or for computation of more accurate results.

Optimization of the C code by hand for the GPU hardware is, however, a tedious, and error-prone task, which is typically beyond the capability of the average programmer. While we implement our optimizations by hand, the main goal of this work is to investigate which optimizations are the most useful and, more importantly, whether the integration of these optimizations into a compiler is possible.

Our approach is to first apply dependency analysis in order to unveil extra parallelism, which we then maximize using high-level source-code transformations, such as loop interchange and loop distribution. Then we translate this code into a basic/naive GPU version written in OpenCL. Then, we implement low-level optimizations related to locality of reference, e.g., memory coalescing, and memory-level optimizations such as placing data in the proper memory layer, e.g., constant, local, or global memory. Finally, we measure and compare the running time of the optimized-GPU, basic-GPU, and sequential-CPU versions of the code.

The empirical evaluation reveals that:

- increasing the degree of parallelism via loop interchange and distribution is fundamental for efficient execution on GPU hardware,
- optimizations related to locality of reference result in speedups as high as 24X with respect to the basic GPU version of the code, and
- memory coalescing is significantly more profitable than the careful placement of data in the most suitable memory segment.

Section 2 explains in detail three code transformations, along with the required data-dependency analysis on arrays, and gives an overview of the

financial program. Section 3 presents the application of loop interchange and loop distribution to the financial application in order to optimize the degree of parallelism. Section 4 and 5 cover memory coalescing and the memory-level optimizations, respectively. In Section 6 we benchmark the different versions of the code in order to assess the importance of the optimizations.

2 Background

The optimization process of a program starts by analysing the source code, in order to determine which code transformations are possible. In Section 2.1-2.3, we introduce the dependency analysis which we use to proof safe code transformations. Section 2.4 and 2.5 present the loop interchange and loop distribution transformations, respectively, and detail on when it is safe and profitable to apply them. The analysis and transformations follow the work in [3]. Finally, in Section 2.6 we explain the financial program on which we intend to apply the transformations.

2.1 Introduction to (Data) Dependency Analysis

A code transformation is *safe* if the original and the transformed code always give the same results, i.e., the semantics of the original program is preserved. One class of transformations, such as loop interchange and distribution, reorganize the loops of the original program but leave the other statements unchanged, albeit their execution order differs in the transformed program.

For this class, the safety of the transformation comes down to proving that the *data dependencies* of the original program are preserved in the transformed code. A data dependency between two instruction mandates that the first instruction must execute and finish before the second instruction can begin executing. Therefore it is not allowed to execute such two instructions out of order.

There are three different types of data dependencies that we will encounter when analysing programs. The first one is the true or flow dependency which occurs when a memory position is first written and then read in a later instruction:

```
A[0] =  ...
...  =  A[0]
```

Therefore this data dependency is also called a read-after-write dependency. In the examples we are using an array reference with a constant subscript for clarity, but the dependencies would still be there if we just have a variable

or an array reference with a variable subscript. For the correctness of the program, this dependency therefore forces the instruction that writes to `A` to be completed before the instruction reading `A` can begin, otherwise the latter instruction will read an incorrect value of `A`.

The next data dependency is the anti-dependency which occurs when a memory position is first read and then written:

```
...   =  A[0]
A[0]  =  ...
```

This data dependency is therefore also called a write-after-read dependency. The crucial part here for correctness is that the correct value of `A` must be read before it is overwritten by some other value in a later instruction.

The last data dependency is the output dependency which occurs when a memory position is first written and then written again in a later instruction:

```
A[0]  =  ...
A[0]  =  ...
```

This data dependency is also called a write-after-write dependency. In order to understand the importance of this dependency, we consider an example:

```
A[0]  =  1;
A[0]  =  2;
B[0]  =  A[0] + 5;
```

If we execute the first two instructions above out of order, then either `1` or `2` may be saved in `A[0]`. For correctness, the first instruction must complete execution before we execute the second instruction to save the correct value of `2` in `A[0]`.

When performing any code transformations it is important that we do not break any of the data dependencies, otherwise the program will be incorrectly transformed. This is stated in the theorem [3] below:

Theorem 1. *Any reordering transformation that preserves every dependency in a program preserves the meaning of that program.*

In the case of loop parallelization, i.e., where one iteration is executed sequentially, but different iterations may be executed out of order, we classify dependencies into *loop-independent* and *loop-carried* dependencies, depending on whether the two instructions that generate the dependency belong

to the same or different iterations, respectively. The safety of the loop parallelization comes down to proving that the loop-carried iterations are preserved (because the loop-independent ones are preserved by default since an iteration is executed sequentially).

2.2 Direction Vectors: Summarizing Dependencies in a Loop Nest

We first set some notations and terminology.

Definition 1. *An* n-level perfect loop nest *is a loop nest in which all statements other than loops appear inside the* n^{th} *(innermost) loop.*

An example of a 2-level perfect-loop nest is shown in Listing 1. Only the innermost loop may contain code that reads or writes values and does calculations. All the outer loops may *only* contain a single `for`-loop construct.

Listing 1: A perfect loop nest used as an example for our dependency analysis.

```
for(unsigned k=1;k<OUTER_LOOP;++k) {
  for(unsigned i=1;i<numX;++i) {
S1  A[k][i] = B[k  ][i-1] + C;
S2  B[k][i] = A[k-1][i-1] + D;
  }
}
```

Iterations inside a perfect loop nest are ordered *lexicographically*, i.e., in an order that satisfies their occurrence in the sequential execution of the loop. For example, `(k=2, i=4)` $<$ `(k=4, i=2)` because the outermost loop iteration `k=2` happens before the outermost loop iteration `k=4`. Similarly, `(k=4, i=2)` $<$ `(k=4, i=4)`, i.e., the inner loop discriminates the order when all the iterations of the outer loops are equal.

Informally, in order for a dependency to exist between two statements S_1 and S_2 executed on two iterations of a perfect loop nest, both statements need to access the same memory location and one of the accesses needs to be a write access, because a read-after-read is not a dependence. The theorem [3] below formally defines a dependence relation in a perfect loop nest:

Theorem 2. *There exists a dependence from statement* S_1 *to statement* S_2 *in a perfect loop nest* if and only if *there exist loop-nest iterations* i *and* j*, with* $i < j$ *or* $i = j$ *according to the lexicographical ordering and there exists a path from* S_1 *to* S_2 *such that:*

- *S_1 accesses memory location M on iteration i, and*
- *S_2 accesses memory location M on iteration j, and*
- *one of these accesses is a write access.*

Dependencies are represented with a directed edge from the statement that executes first in the loop nest, named the source*, i.e., S_1 executed on iteration i, to the one that executes later, named* the sink *of the dependence, i.e., S_2 executed on iteration j.*

We illustrate first how to find dependencies in the code in Listing 1. It is important to note that the following methods only work when used on perfectly nested loops. Dependencies for `A` can be found using equations of the subscripts of each dimension of the array. Let us first take a closer look at the array `A` in Listing 1.

We isolate the first and second subscripts of `A` in statements $\mathtt{S}_1$ and $\mathtt{S}_2$, and distinguish any index used more than once with a subscript. We set them equal to each other and see that

$$i_1 = i_2 - 1 \implies i_1 < i_2$$
$$k_1 = k_2 - 1 \implies k_1 < k_2$$

From this we conclude that there is no loop-independent dependency since a value is written to and read from different memory positions in array `A`. Next we check for loop-carried dependencies. It is essential to determine any loop-carried dependencies in a perfect loop nest in order to perform the loop transformations that we present in the following sections. In particular, many transformations can only be done when there are no loop-carried dependencies. Even if there are loop-carried dependencies, some loop transformations may still be applicable, so in any case, we need to know which dependencies are present. We first start with some definitions.

Definition 2. *Assuming that, using the method presented above, we have found the relation between i_1 and i_2 for the index i used in two array subscripts, to the same array. Then the* dependency direction*, denoted $d(i)$ is defined as follows:*

$$d(i) = \begin{cases} \texttt{'<'} & \text{if } i_1 < i_2 \\ \texttt{'='} & \text{if } i_1 = i_2 \\ \texttt{'>'} & \text{if } i_1 > i_2 \end{cases}$$

A commonly used (and very useful) concept for summarizing the dependencies that occur between a pair of statements involving array accesses in a perfect-loop nest is the *dependency direction vector.*

Definition 3. *A dependency direction vector for a pair of accesses to an n-D array A with the indices $j_1, j_2, ..., j_n$ within an n-level perfect loop nest is a tuple that models the direction for each index according to definition 2, that is, the tuple is given as*

$$(d(j_1), d(j_2), ..., d(j_n))$$

How to find a dependency direction vector is best understood through an example. In the example of Listing 1, we have an access to array `A` in statement $\mathtt{S}_1$ and another one in $\mathtt{S}_2$. We have also found that for `A`'s two indexes, k and i, there exist the dependency directions "<" and "<", respectively.

Using definition 2 and 3 we find the dependency direction vector for the pair of accesses to `A` to be $(<, <)$. Similarly one can derive the dependency direction vector for the `B` array to be $(=, <)$. Intuitively one may think that the direction vectors "<" and ">" give rise to loop-carried dependencies, while "=" does not.

2.3 Using Direction Vectors to determine safe parallelization

Our goal is to derive if a loop has any loop-carried dependency. If we find that it does not, then we can safely parallelize the loop. The following theorem gives notion as to when a loop is parallel.

Theorem 3. *A loop, and in particular an inner loop, in a perfect loop nest is parallel if and only if for all its non-"=" symbols in its direction vectors, there exists an outer symbol that is "<".*

This also implies that a loop with only "=" symbols in its corresponding column of the direction vectors is parallel.

It is important to consider all direction vectors within a perfect loop nest in order to determine whether any of the loops are parallel. Let us use this theorem on the running example. We have the following two direction vectors:

$$\mathtt{A}\text{: } (< <)$$
$$\mathtt{B}\text{: } (= <)$$

The first column corresponds to the outermost loop. Let us see if we can parallelize it. For array `A` we have the symbol "<", which does not have an outer symbol "<", since it is already the outermost symbol. For array

`B` we have no dependency. Hence, array `A` has a loop-carried dependency making the outermost loop sequential. The second column corresponds to the innermost loop. For array `A` we have the symbol "<", but we also have an outer symbol that is "<", hence the loop has no dependency regarding `A` according to the theorem. For array `B` we have a "<", but the outer symbol is not "<", it is "=". Therefore there is a loop-carried dependency on `B` making the innermost loop sequential. If there were no array references to array `B` in the loop nest, then we would have been able to parallelize the innermost loop.

In total then, none of the loops can be executed in parallel, since a dependency on `A` makes the outermost loop unsafe for parallelization, while a dependency on `B` makes the innermost loop unsafe for parallelization.

2.4 Loop Interchange

The code in Listing 2 shows one of the important code transformations that we will make use of, namely loop interchange. The theorem [3] below states when loop interchange is safe.

Theorem 4. *Let T be a transformation that is applied to a loop nest and that does not rearrange the statements in the body of the loop. Then the transformation is safe if, after it is applied, none of the direction vectors for dependencies with a source and a sink in the nest has a leftmost non-"=" symbol that is ">".*

This theorem tells us, that when we want to interchange two loops, then we must also interchange the columns of the direction vectors in the same way that we plan to interchange the loops. This must not result in any direction vector that has the direction ">" as its leftmost non-"=" symbol. Otherwise, intuitively, in the transformed code an iteration depends on the result of some *future* iteration, which is clearly impossible.

We have seen in the previous section that none of the loops in Listing 1 can be directly parallelized. We will try to interchange the two loops in order to make any of the loops parallel. First, we will use the theorem to check if loop interchange is safe. After interchanging the columns of the direction vectors we have:

$$\texttt{A}\text{: } (< <)$$
$$\texttt{B}\text{: } (< =)$$

Now we apply Theorem 4 to see if this is safe. Since none of the direction vectors has a leftmost non-"=" symbol that is ">", interchanging the two

loops is safe. At this point we can once again go through the analysis of Section 2.3 and apply Theorem 3. This tells us that the innermost loop is now parallel because the outer loop carries all dependencies, i.e., on both `A` and `B`. For example there is now a loop-carried dependency on `B` for the outermost loop, but no dependency for either array in the innermost loop, which therefore can be safely executed in parallel. The loop-interchange transformation is performed in Listing 2 on the code from Listing 1.

Listing 2: Loop interchange that increases the degree of parallelism. No loops on the left-hand side can be parallelized, while the innermost loop on the right-hand side can be executed in parallel.

```
for(unsigned k=1;k<OUTER_LOOP;++k) {
  for(unsigned i=1;i<numX;++i) {
    A[k][i] = B[k  ][i-1] + C;        ==>>
    B[k][i] = A[k-1][i-1] + D;
  }
}
```

```
for(unsigned i=1;i<numX;++i) {
  for(unsigned k=1;k<OUTER_LOOP;++k) {
    A[k][i] = B[k  ][i-1] + C;
    B[k][i] = A[k-1][i-1] + D;
  }
}
```

Listing 3: Loop interchange that increases the granularity by moving the `k` loop, which is parallel, to the outermost position. On the right-hand side, the accesses to the array `A` is furthermore cached efficiently.

```
for(unsigned i=1;i<numX;++i) {
  for(unsigned k=0;k<OUTER_LOOP;++k) {
    A[k][i] = A[k][i-1] + C;          ==>>
    ...
  }
}
```

```
for(unsigned k=0;k<OUTER_LOOP;++k) {
  for(unsigned i=1;i<numX;++i) {
    A[k][i] = A[k][i-1] + C;
    ...
  }
}
```

The loop interchange transformation is important for a number of reasons. First, by interchanging two loops we change the order of the accesses to the arrays so that a previously sequential loop may now be executed in parallel, thereby increasing the degree of parallelism of the program. This is what happens in Listing 2 where loop interchange allows parallel execution of the right-hand-side `k`-loop.

Secondly, it can improve the granularity in a program by either making the granularity coarser or finer. By performing a similar analysis as above, one can verify that, in Listing 3, loop `k` is parallel, i.e., by the direction vector $(<, =)$ of array `A` and that loop `i` is sequential, and also that loop interchange is safe. By interchanging the loops, as done on the right-hand side, each thread of execution will execute a larger piece of code making the granularity coarser. At the same time the parallel loop becomes the outermost loop, which is preferable for a number of reasons; for instance to reduce the number of synchronizations between threads. The transformation can be, if desired, reversed to make the granularity finer. Finally, loop interchange may change the order in which the elements of an array are accessed, for example to

maximize the locality of reference. The resulting code, depicted in Listing 3, exhibits better (overall) cache usage.

2.5 Loop Distribution

We will also make use of a transformation called loop distribution. Loop distribution, as shown in Listing 5, distributes the outermost loop across the inner loops resulting in new loop nests. We use this transformation to increase the degree of parallelism of a program.

An important transformation often used in connection with loop distribution is *privatization*, which we use to remove output dependencies from a loop nest. Such output dependencies would occur in Listing 4 if we were to parallelize the outermost loop, because each thread would be writing different values to the same position in `A` and `B`. The privatization transformation creates a private copy of `A` and `B` for each thread and then makes sure that the correct copy is used when sequential execution continues. For a GPU parallelization this means performing array expansion on both arrays, such that we create an extra outer dimension of size equal to the length of loop `k`.

Listing 4: Applying array expansion to the left-hand side to remove all output dependencies

```
for(k=1;k<OUTER_LOOP;++k) {
  for(i=1;i<numX;++i) {
    A[i] = A[i] * D[k][i] + 1;       ==>>
  }
  for(j=1;j<numY;++j) {
    B[j] = B[j] + A[j] + C[k];
  }
}
```

```
for(k=1;k<OUTER_LOOP;++k) {
  for(i=1;i<numX;++i) {
    A[k][i] = A[k][i] * D[k][i] + 1;
  }
  for(j=1;j<numY;++j) {
    B[k][j] = B[k][j] + A[k][j] + C[k];
  }
}
```

The array expansion transformation is always safe, but it is made at the cost of extra memory usage, hence it should not be used unless it enables transformations such as this which in turn makes parallelization applicable.

Loop distribution can be applied when there is no cycle of dependencies between the instructions of the inner loops. Looking at the code in Listing 5 we see that there is a dependency on array `A` from loop `i` to loop `j`, because the value written in array `A` in loop `i` is read from array `A` in loop `j`. There is, however, not a dependency from the array reference to array `A` in loop `j` back to loop `i`. Therefore there is not a cycle of dependency and loop distribution can safely be performed.

In the example in Listing 5 on the left-hand side, we see that all the loops are parallel, but it is not a perfectly nested loop, which means that the parallelism is not nested. Such unnested parallelism could be exploited on

the CPU, but it would require some type of dynamic scheduling of the work load. Furthermore, the parallelism in the `k`-loop would probably be sufficient for the CPU and running the `i`- and `j`-loops sequentially would therefore be advantageous, since it reduces scheduling and synchronization overhead. This type of dynamic scheduling is not possible on the GPU, and hence, this forces sequential execution on the otherwise parallel `i`- and `j`-loops. The problem is that the loops are not nested in a perfect loop nest.

Listing 5: Applying loop distribution to the left-hand side. All loops are parallel, but the right-hand-side loops exhibit nested parallelism.

```
for(k=1;k<OUTER_LOOP;++k) {
  for(i=1;i<numX;++i) {
    A[k][i] = A[k][i] * D[k][i] + 1;     ==>>
  }
  for(j=1;j<numY;++j) {
    B[k][j] = B[k][j] + A[k][j] + C[k];
  }
}
```

```
for(k=1;k<OUTER_LOOP;++k) {
  for(i=1;i<numX;++i) {
    A[k][i] = A[k][i] * D[k][i] + 1;
  }
}
for(k=1;k<OUTER_LOOP;++k) {
  for(j=1;j<numY;++j) {
    B[k][j] = B[k][j] + A[k][j] + C[k];
  }
}
```

The application of loop distribution turns the left-hand side into two perfectly nested loops, which in turn results in nested parallelism. This transformation enables us to exploit all the parallelism present in the loops on the GPU.

Loop distribution can be used to enable a higher degree of parallelism in many other situations, such as when the outer loop contains larger loop nests in which some of the loops are sequential, or to split a sequential loop into multiple parallel loops.

2.6 The Financial Program

This section is meant to give a brief overview of the program, in order to get a sense of the type of application that we are optimizing. It does not include all parts of the financial program, e.g., we do not give the boundary conditions.

The program we are examining is a real-world implementation that solves a second-order partial differential equation (PDE) of the form

$$\frac{\partial f}{\partial t} + \sum_{i=1}^{d} \mu_i \frac{\partial f}{\partial S_i} + \frac{1}{2} \sum_{i,j=1}^{d} \sigma_i^2 \frac{\partial^2 f}{\partial S_i \partial S_j} - rf = 0 \tag{1}$$

The unknown f denotes the price of a financial asset and depends on the underlyings $\mathcal{S} = (S_1, ..., S_d) \in [0, \infty)^d$. For our program, the number of dimensions d is 2 and hence the function f and the variables μ_i, σ_i and r

depends on $(x, y, t) \in \mathcal{S} \times [0, T]$. We have the terminal condition $f(x, y, T) = F(x, y), (x, y) \in \mathcal{S}$, where $F : \mathcal{S} \to \mathbb{R}$ is the payoff function of the financial asset that we wish to price and it is known beforehand. More details on the model can be found in [6].

In many cases, this PDE does not have a closed-form solution. Therefore we will discretize the model in order to find a numerical solution. We will start by discretizing the model in the space dimension. After forming an equidistant discretization mesh, we approximate the partial derivatives by finite differences. We approximate the first-order partial derivatives by

$$\frac{\partial f(x, y, t)}{\partial x} = \frac{f^t_{x+1,y} - f^t_{x-1,y}}{2\Delta x}, \tag{2}$$

where $f^t_{x,y}$ denotes the value of f at point (x, y) at time t. We approximate $\frac{\partial f(x,y,t)}{\partial y}$ similarly. We approximate the second-order derivatives by the central difference

$$\frac{\partial f(x, y, t)}{\partial x^2} = \frac{f^t_{x+1,y} - 2f^t_{x,y} + f^t_{x-1,y}}{(\Delta x)^2}, \tag{3}$$

which we also define for $\frac{\partial f(x,y,t)}{\partial y^2}$ and $\frac{\partial f(x,y,t)}{\partial y \partial x}$. For approximating the solution of the PDE in the time dimension, we will use the Crank-Nicolson scheme.

The Crank-Nicolson scheme makes use of the explicit and implicit Euler schemes, both of which we can derive by approximating the partial derivatives in time by the forward difference for the explicit Euler

$$\frac{\partial f(x, y, t)}{\partial t} = \frac{f^{t+1}_{x,y} - f^t_{x,y}}{\Delta t} \tag{4}$$

and the backward difference for the implicit Euler

$$\frac{\partial f(x, y, t)}{\partial t} = \frac{f^{t-1}_{x,y} - f^t_{x,y}}{\Delta t} \tag{5}$$

The Crank-Nicolson scheme then defines the approximate solution at time step $t = t - \Delta t$ by performing both the explicit and implicit Euler schemes and taking the average.

We find an approximate solution to the PDE in Eq. (1) by plugging in the approximated partial derivatives from Eq. (2)-(5) into Eq. (1) and solving for the next time step. The end result is a two-dimensional five-point stencil Jacobi iteration, which we apply in each time step. We start at the end of the time interval, i.e. with $f^T_{x,y} = F(x, y), (x, y) \in (S)$ and calculate steps backwards in time until we reach the starting time $t = 0$.

Calculating the explicit Euler step is straightforward, but the implicit Euler step requires a solution to a system of linear equations in each time

step. As the program is solving the PDE for $d = 2$ dimensions, this yields a coefficient matrix with a tridiagonal structure which we solve efficiently using the standard tridiagonal matrix solver. See [6] for a short overview the tridiagonal matrix solver.

In short, the program solves a common partial differential equation in finance which can be used for such things as market calibration and financial asset pricing. A solution to the PDE is approximated using finite differences, which makes it a memory-bound algorithm. In the following we shall examine the feasibility of parallelizing such an (memory-bound) algorithm on GPUs.

3 Parallelizing Crank-Nicolson for GPU execution

The Crank-Nicolson method already exhibits some degree of parallelism that might be enough for parallelization in a multi-core CPU environment. Since the GPU architecture excels at executing massively parallel applications, we aim to maximize the degree of parallelism in the program to make full use of the GPU hardware.

We increase the degree of parallelism by applying the dependency analysis and code transformations discussed in Section 2. Below is a simplified version of the code which we use as the working example. We have found that some of the loops does not allow for parallel execution, and we annotate these as sequential. At this point, the other loops are also sequential due to output dependencies, but we will try to make them parallel.

Listing 6: A simplification of our application that we use as the working example. Sequential loops are annotated.

```
for(k=0; k<OUTER_LOOP; ++k) {
    for(g=numT; g>=0; --g) { // Sequential
        for(jj=0; jj<numY; jj++) {
            for(i=0; i<numX; i++) { // Sequential
                u[jj][i] = B1 + time[g]*C1*myResult[i][jj];
            }
            tridag(numX,u,u);
        }
        for(ii=0; ii<numX; ii++) {
            for(j=0; j<numY; j++) { // Sequential
                y[j] = time[g]*u[j][ii] - C2;
            }
            tridag(numY,y,myResult);
        }
    }
}
```

We start be applying array expansion to eliminate the output dependencies that would emerge when parallelizing loop `k`, loop `jj` and loop `ii`. This

results in the following program:

Listing 7: Applying array expansion to array `u`, `myResult` and `y` from Listing 6.

```
for(k=0; k<OUTER_LOOP; ++k) {
    for(g=numT; g>=0; --g) { // Sequential
        for(jj=0; jj<numY; jj++) {
            for(i=0; i<numX; i++) { // Sequential
                u[k][jj][i] = B1 + time[g]*C1*myResult[k][i][jj];
            }
            tridag(numX,u,u);
        }
        for(ii=0; ii<numX; ii++) {
            for(j=0; j<numY; j++) { // Sequential
                y[k][ii][j] = time[g]*u[k][j][ii] - C2;
            }
            tridag(numY,y,myResult);
        }
    }
}
```

Through our dependency analysis we find that parallel execution of loop `k`, loop `jj` and loop `ii` is now safe. The `tridag` function is run sequentially by choice: While parallelizing `tridag` is possible, it requires a significant algorithmic change that is beyond the ability of the compiler. Furthermore, abundant parallelism can be found elsewhere making sequential execution the most efficient option.

We now apply the loop interchange transformation to interchange loop `k` with loop `g` as shown in Listing 8. This gets us closer to exploiting the nested parallelism of loop `k` and the `jj`- and `ii`-loops, and it is safe to interchange the two loops since loop `k` has no loop-carried dependencies.

Listing 8: Interchange of loop `k` with loop `g`.

```
for(g=numT; g>=0; --g) { // Sequential
    for(k=0; k<OUTER_LOOP; ++k) { // Parallel
        for(jj=0; jj<numY; jj++) { // Parallel
            for(i=0; i<numX; i++) { // Sequential
                u[k][jj][i] = B1 + time[g]*C1*myResult[k][i][jj];
            }
            tridag(numX,u,u);
        }
        for(ii=0; ii<numX; ii++) { // Parallel
            for(j=0; j<numY; j++) { // Sequential
                y[k][ii][j] = time[g]*u[k][j][ii] - C2;
            }
            tridag(numY,y,myResult);
        }
    }
}
```

While it would be possible to run the `jj`- and `ii`-loops in parallel as well, this would require some type of dynamic scheduling of the work load, which is not possible on the GPU, cf. Section 2.5. Hence, this forces sequential execution on the otherwise parallel `jj`- and `ii`-loops. The problem

is that the loops are not nested in a perfect loop nest.

To regain this parallelism, the next step is to use the loop distribution transformation to distribute loop `k` across the `jj`- and `ii`-loops, which is shown in Listing 9. The result is two inner perfect loop nests inside the sequential `g`-loop, both of which exploit the nested parallelism of two parallel loops and, hence, we are now able use the full amount of parallelism present in the program.

Listing 9: After distributing loop `k` over the inner loop nests.

```
for(g=numT; g>=0; --g) { // Sequential
    for(k=0; k<OUTER_LOOP; ++k) { // Parallel
        for(jj=0; jj<numY; j++) { // Parallel
            for(i=0; i<numX; i++) { // Sequential
                u[k][jj][i] = B1 + time[g]*C1*myResult[k][i][jj];
            }
            tridag(numX,u,u);
        }
    }
    for(k=0; k<OUTER_LOOP; ++k) { // Parallel
        for(ii=0; ii<numX; ii++) { // Parallel
            for(j=0; j<numY; j++) { // Sequential
                y[k][ii][j] = time[g]*u[k][j][ii] - C2;
            }
            tridag(numY,y,myResult);
        }
    }
}
```

The two inner perfect loop nests translate nicely into two kernels for the GPU. It is necessary to use two different kernels here because the loop nests are of different size: The first one is of size `OUTER_LOOP`$*$`numY` and the other of size `OUTER_LOOP`$*$`numX`. Also, while not apparent from the example code, the loop interchange transformation improved the locality of reference in the code. Since OpenCL only supports one dimensional arrays, we will need to flatten the indexes of the arrays. The basic GPU version that we refer to later is the code above rewritten into two kernels in the OpenCL programming language [4]. The `g`-loop is run sequentially on the CPU which then invokes the two OpenCL kernels in each iteration.

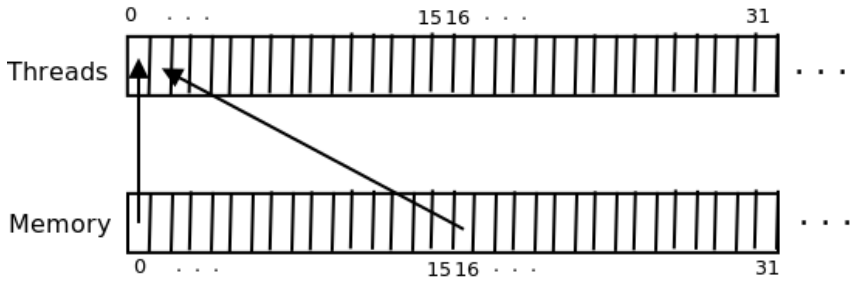
4 Memory Coalescing by Transposition

Code that is written for the CPU architecture does typically not execute fast on the GPU, since it is not tailored for some distinct features of the GPU architecture, which it needs to be in order to get the desired speed out of the GPU. One such feature is when accessing data from global memory.

When a memory location is accessed by a single thread in a work group that is executing on the GPU, an entire block of data from the memory is

always accessed. It is usual that the block of data is sixteen words. If not all sixteen words of this block are used, then we have an uncoalesced memory access pattern as shown in Figure 1.

Figure 1



In figure 1 we observe the worst-case scenario, where the first sixteen words are accessed, but only a single word from the transferred block is used by a thread, here by thread `0`. The next thread also accesses a full block and only uses one value from it. Situations such as this is common when the thread ID is not used to index the innermost dimension of an array.

Let us consider the example below. The dimensions of the array `y` is `[16][16]`.

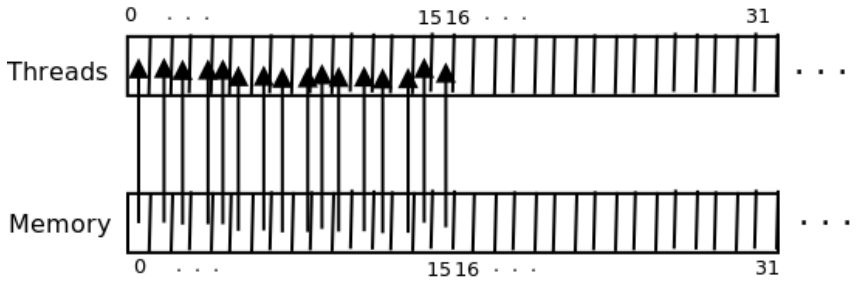
Listing 10: Example array reference on the GPU that results in uncoalesced memory accesses.

```
int i = get_global_id(0);
for(j=0; j < 16; j++) {
  ... = y[i * 16 + j];
}
```

On the GPU, every thread in a work group always executes the same instruction. For the first iteration of the `j` loop, the first thread, with thread ID 0, reads `y[0*16 + 0]`. The second thread reads `y[1*16 + 0]`. The third thread reads `y[2*16 + 0]`. In order for 16 threads to get the data that they need, 16 memory reads are performed. This is exactly the pattern of uncoalesced memory access described above. The subscript in `y` has a stride equal to the size of the innermost dimension of `y`.

The optimal memory access pattern on the GPU is to have a stride of one as shown in figure 2.

Figure 2



Here we access one block of data in which each thread uses exactly the one value from the block that they need. Such one-strided access is called a coalesced memory access pattern and it is the pattern of choice for the GPU. In this simple example, we have just reduced the number of memory transfers by a factor of 16. In summary then, if 16 consecutive threads access in an (single) instruction 16 consecutive memory locations, then this requires only one memory transfer, otherwise potentially up to 16 memory transfers.

We can achieve such memory coalesced access simply by transposing the subscripts of the array `y`. This change is similar to changing the memory layout of array `y` from row-major to column-major. Let us again look at the example:

Listing 11: Example array reference on the GPU, now transposed, which results in coalesced memory accesses.

```
int i = get_global_id(0);
for(j=0; j < 16; j++) {
  ... = y[j * 16 + i];
}
```

For the first iteration of the `j` loop, the first thread, with thread ID 0, reads `y[0*16 + 0]`. The second thread reads `y[0*16 + 1]`. The third thread reads `y[0*16 + 2]`. Clearly, we have memory coalesced access, but we are now reading different data than before. For this transformation to be correct, we also need to transpose the data in array `y`.

However, changes in the data layout needs to be performed uniformly throughout the code. With our program, solving coalescing in one place is breaking coalescing in another place. For the first kernel we would like to have array `u` transposed, which breaks the already coalesced memory access to array `u` in the second kernel. The reverse situation applies to the `myResult` array. Performing matrix transposition between the two kernels at run-time solves this problem. The result is the following four kernels.

```
// Kernel #1
int j = get_global_id(0);
```

```
int k = get_global_id(1);
for(i=0; i<numX; i++) {
  idxB = j + numY*(i+numX*k)
  u[idxB] = B_1 + time[g]*C_1*myResult[idxB];
}
tridag(numX,u,u);

// Kernel #2
Transpose(u)

// Kernel #3
int i = get_global_id(0);
int k = get_global_id(1);
for(j=0; j<numY; j++) {
  idxA = i + numX*(j+numY*k)
  y[idxA] = time[g]*u[idxA] - C_2;
}
tridag(numY,y,myResult);

// Kernel #4
Transpose(myResult)
```

5 Memory-level Optimizations

The GPU features several levels of memory: constant memory, private memory, local memory, image memory and global memory. Naturally, some memory levels are faster than others, but it is not transparent which one is used; the programmer needs to specify within the code which data is stored in which memory level. In the following sections we will present optimizations that use the constant and image memory segments as presented in [1].

5.1 Constant Memory

It it suitable to place data in the constant memory segment when the program exhibits a certain form of access pattern. When the same element of data is read by all threads in a work group in the same instruction and if the data is read-only, then the best placement is in the GPU constant memory segment. Data in the constant memory segment will be cached which lowers the total amount of memory transfers needed.

The cache on the GPU is, however, very small and multiple work groups share the same cache. Therefore, only a very small amount of cache, is available for each work group and it is also possible for other work groups to evict the cache line of another work group. It is therefore important to structure data such that its layout matches the sequential order in which the data is read by the kernel code. For example, if we know which eight elements of data will be read in a given iteration, then it is essential that the

data is structured such that all the eight elements of data is located on the same cache line and not on multiple cache lines.

Let us look at some code inside our kernel that uses data which is suitable for placing in constant memory. In the code below, all threads in a work group will read the same values of the array `Dx`, since the subscript is based solely on the loop index variable and not on a thread ID.

Listing 12: We place `Dx` in constant memory for cached memory access by changing its identifier to `__constant`.

```
__kernel void C_Kernel(__global float* Res, __constant float* Dx) {
  for(i=0;i<numX;i++) {
    ...
    idx = ...
    tmp += 0.5*C1*Dx[i*3 + 0]*Res[idx];
    tmp += 0.5*C2*Dx[i*3 + 1]*Res[idx];
    tmp += 0.5*C2*Dx[i*3 + 2]*Res[idx];
    ...
  }
}
```

In the first iteration of the loop, we are reading from `Dx[0]`, `Dx[1]` and `Dx[2]`. This means that data is read by all the threads of a work group in a stride-one fashion. Hence, we must structure the data of `Dx` in the same way as shown in Figure 3.

Figure 3

This type of data layout is called array of structs in the literature and is the optimal data layout for the constant memory [5]. When a cache line is read, it contains all the three elements of data needed in a loop iteration. If the cache line is still in the cache in the next iteration then, depending on the cache line size, the next three elements of data will already be in cache. We place the array in constant memory by setting the memory identifier in front of the array in the argument list to `__constant` as shown in Listing 12.

5.2 Image Memory

If each thread needs to read three or more consecutive elements of data in each iteration of the loop, then one should consider converting the array to a `float4` array and accessing four elements of data at once. To do this, the data also needs to be read-only, and the data needs to be placed in the read-only image memory which in many cases improves the speed at which data is loaded.

Data read from the image memory is cached which makes it faster. The layout of data in the image memory is, however, opaque to the user. It is determined by the OpenCL implementation, but the layout is usually chosen to optimize locality of reference, for instance such as the Morton ordering does [2].

The example code in Listing 13 is taken from one of the kernels that we are executing. On the left-hand side we see that three consecutive elements of data is referenced by each thread in the work group, i.e. `myDy[idx]`, `myDy[idx+1]` and `myDy[idx+2]`. Instead of reading three elements of data from the global memory, we can read one element of the `float4` type from the image memory containing the three elements. More importantly, the data is read in a three-strided access pattern which results in uncoalesced memory access. The use of `float4` and the image memory also solves the coalescing issues.

Listing 13: On the left we are reading `myDy` from global memory. On the right, we place `myDy` in image memory and read four elements at a time using the function `read_imagef()`.

```
__kernel void I_Kernel(
  __global float* Res,
  __global float* myDy) {
  for(i=0;i<numX;i++) {
    ...
    idx = get_global_id(0)*3 + ...
    idx2 = ...
    tmp4.x = 0.5*C*myDy[idx  ]*Res[idx2-1];
    tmp4.y = 0.5*C*myDy[idx+1]*Res[idx2];
    tmp4.z = 0.5*C*myDy[idx+2]*Res[idx2+1];
    ...
  }
}
```

```
__kernel void I_Kernel(
  __global float* Res,
  __read_only image2d_t myDy) {
  for(i=0;i<numX;i++) {
    ...
    idx = get_global_id(0) + ...;
    idx2 = ...
    temp = read_imagef(myDy, ..., (idx,0));
    tmp4.x = 0.5*C*temp.x*Res[idx2-1];
    tmp4.y = 0.5*C*temp.y*Res[idx2];
    tmp4.z = 0.5*C*temp.z*Res[idx2+1];
    ...
  }
}
```

As with the constant memory optimization, we also need to change the memory identifier and we use a function call, `read_imagef()`, to read data from the image memory. We also need to restructure the array `myDy` from an array of structs of size three to an array of structs of size four, to account for the extra space used by the `float4` type. Furthermore, the host code needs to create an image buffer on the GPU to which the content of `myDy` is transferred to.

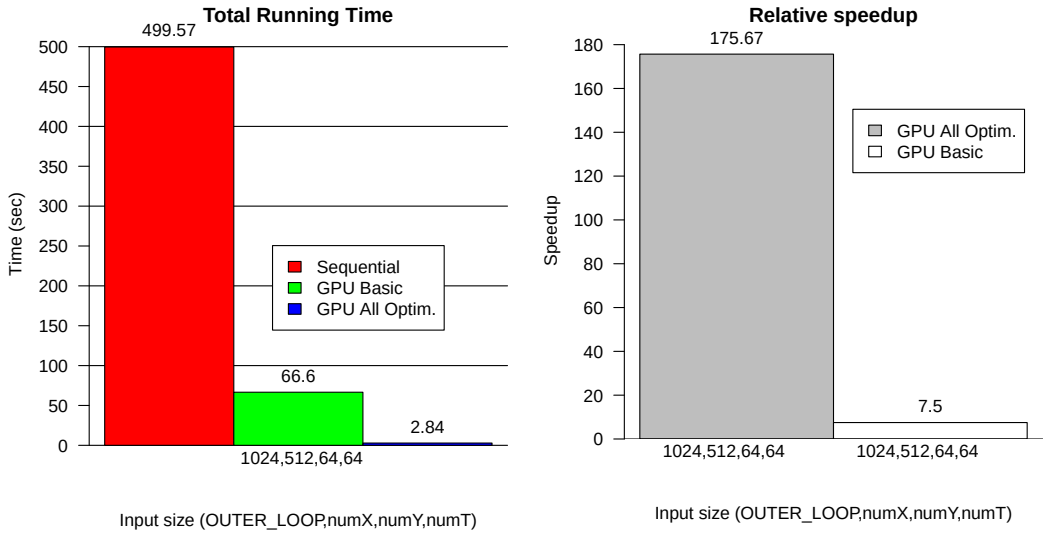
These are all the optimizations that we apply. While many others can be made, such as blocking the code to exploit the local memory segment, these were not performed, since they are slightly more complex for the compiler to make.

6 Benchmark

In this section we assess by how much our transformations and optimizations actually made the code faster. The benchmark was executed on a machine with a Intel Quad core CPU at 2.4GHz and 4MB L2 cache along with the Nvidia GTX690 GPU. The code was executed using single-precision floats and the GPU has 3090 GFLOPS in single precision. The CPU code is totally unoptimized, i.e. single threaded with no use of SIMD, and we only include it to have a naive baseline of the running time on the CPU to compare against.

In total we have three versions that we will benchmark: the CPU version, the naive GPU version and the optimized GPU version. We executed all three versions ten times each and computed the average running time. All versions were compiled only with the O3 flag with gcc and the Nvidia OpenCL compiler. In Figure 4, we show the total running time in seconds for the different versions of the code and the relative speedup of the GPU versions in comparison to the sequential version. We benchmarked the code using different work-group sizes and we found 64 to be the best work-group size overall.

Figure 4: Benchmark of the basic and optimized GPU versions on the Nvidia GTX690. *Left*: Comparison of the total running time of the sequential C code versus the GPU versions. *Right*: The speedup of the GPU versions relative to the C code.

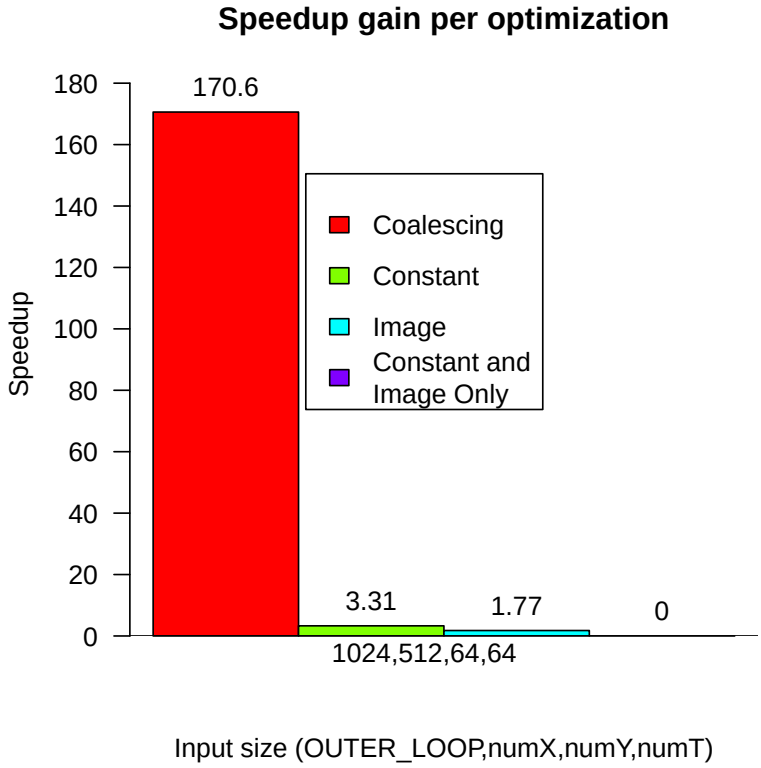


The last input argument, `numT`, is the number of time steps that we execute, cf. Section 2.6.

The graph to the right shows that the basic GPU version with no memory-level optimizations is only 7.5X faster than the sequential CPU code. If we had a fully optimized CPU code running on a CPU with the same cost as the Nvidia GTX690 card, e.g. the Intel Sandybridge, then the basic GPU version would probably be many times slower than the optimized CPU code. Clearly one cannot just port the CPU code to the GPU and expect to get a massive performance boost just from that. One also needs to perform optimizations to the GPU code. With the optimizations that we have applied, we gained a total of 175X speedup. The optimized GPU version might be quicker than a fully optimized CPU version, but not by a lot. There is still many optimizations that we could perform on the GPU code.

Indeed, we are still only using a fraction of the compute capacity of the GPU, but our goal was to determine which optimizations are the most profitable to do. Then, we can decide which optimizations we should focus on, if we were to implement the optimizations into a compiler. We have measured the speedup gained by each of the three optimizations that we performed. In Figure 5, we show how much each optimization contributed to the total speedup.

Figure 5: Speedup contribution for each of the three optimizations that we performed: Memory coalescing, placing the appropriate data in constant memory, and placing the appropriate data in the image memory. *Red*: Only memory coalescing. *Green*: Adding the constant memory optimization on top of the memory coalescing. *Teal*: Adding the image memory optimization on top of the memory coalescing. *Blue*: Only doing the constant memory and image memory optimizations.



We see that the memory coalescing optimization accounts for almost all of the total speedup. This makes sense when looking at the code, since we have a memory-bound algorithm, we spend most of the time reading or writing to the global memory. The memory coalescing optimization speeds up this process significantly. Adding either the constant memory or image memory optimization on top of the memory coalescing yields only a very minor speedup. From the code we observe that these optimizations only speed up small parts of the total code and we therefore cannot expect that they give a large improvement of the total running time. Also, even with these optimizations, we still have a large amount of accesses to the global memory. The blue bar in Figure 5 shows how much speedup we gain by only performing the constant memory and image memory optimizations without

the memory coalescing. There was no speedup from doing this, in fact, the code became slightly slower. Again, the main problem with this code is the amount of memory accesses to the global memory, which we are not improving much with these two optimizations.

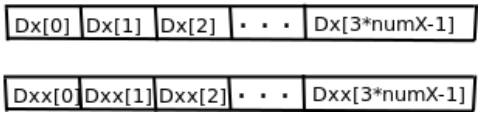
In conclusion we have found that the memory coalescing optimization is definitely the most profitable optimization on GPUs, perhaps in particular when executing memory-bound algorithms. It accounts for 97% of the total speedup. The optimization is particularly important on the Nvidia GPUs. There already exists AMD GPUs, e.g. the AMD FirePro W8000, which implements a hardware unit that makes it just as efficient to do uncoalesced memory accesses. Performing the memory coalescing optimization on a GPU architecture such as this will not improve the running time of an application at all.

We did not benchmark the impact of performing the parallelism-enhancing transformations such as loop interchange and distribution along with privatization. Through the optimization process we found that they were essential, since otherwise we would not have enough parallelism to make program execution on GPU hardware practical, in fact, if we did not do any of them, we would not be able run any loop in parallel safely.

We should also discuss how much effort is needed to perform the three low-level memory-level optimizations. We have already seen that the act of performing them is not very difficult. The question is if we should always perform these optimizations, and in case not, under which circumstances should we not do it. For most programs, memory coalescing can be performed, and it will always give a speedup.

The same goes for the constant memory optimization, but only if it is used properly, that is, if multiple elements data is always being read from the same cache line in our program. If only one word of data is read before the cache line gets evicted, no speedup is gained. As we mentioned earlier, the constant memory is best utilized when the data layout corresponds to the array of structs scheme. Sometimes though, you might have several arrays located in the constant memory: in our program we have two such arrays, `Dx` and `Dxx`. Each is stored in the proper manner in the constant memory with the following layout:

Figure 6

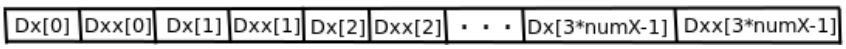


Storing the data in two separate arrays was probably intuitive to the programmer, but it is not optimal for our program, because we can observe from the code that the data is accessed in the following order:

```
Dx[0]; Dxx[0]; Dx[1]; Dxx[1]; Dx[2]; Dxx[2];
```

and so on. One worthwhile optimization is therefore to merge the `Dx` and `Dxx` arrays into one array with the following data layout:

Figure 7



in order to minimize the number of cache misses as much as possible. Additional optimizations such as this will also be needed in order to make full use of the GPU architecture.

It is harder to say if we should always perform the image memory optimization, since it all depends on the cache behaviour.

However, the analysis needed to perform these optimizations should be straightforward; we simply determine if the data is read in the correct pattern for each of the three optimizations. For example, checking if an array should be placed in constant memory might be as simple as checking if a thread ID is present in the subscript of an array reference.

7 Conclusion

In this report we have presented the optimization of a real-world program that solves a partial differential equation used for asset pricing in financial applications. The optimization was carried out by first porting the code to be run on GPU hardware, and then doing three optimizations beneficial to the GPU architecture.

We first introduced the necessary tools: loop transformations needed to extract sufficient parallelism for the GPU and dependency analysis through direction vectors, which we needed to determine if the transformations were safe. Then we performed these transformations on the financial program and ported the code to OpenCL for GPU execution.

Through our benchmarks we found that this code was still not using the GPU hardware effectively. So we discussed three notable features of the GPU architecture that necessitated three optimizations to speed up the code: memory coalescing and using the constant and image memory segments. Overall, we gained a speedup of 175X over the unoptimized CPU

code, while finding that memory coalescing was the most important optimization, accounting for 97% of the total speedup.

Although we performed the optimizations by hand, we found that they were straightforward to do, and for many programs, we think that they would require little analysis. Therefore we find that it is possible to implement a compiler that performs these optimizations automatically with success.

Bibliography

[1] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink, Compiling a high-level language for GPUs:(via language support for architectures and compilers), *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, ACM (2012), 1–12.

[2] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL*, Morgan Kaufmann Publishers Inc. (2011).

[3] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*, Morgan Kaufmann Publishers Inc. (2002).

[4] Khronos Group, OpenCL Specification v1.2r19 (Nov. 2012). Available at `http://www.khronos.org/registry/cl/`.

[5] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann Publishers Inc. (2010).

[6] C. Munk, Introduction to the numerical solution of partial differential equations in finance (2007). Available at `http://mit.econ.au.dk/vip_htm/cmunk/noter/PDENOTE.pdf`.