Programming Massively Parallel Hardware
GROUP PROJECT

Olugbenga Oti
qlg389

Ehsan Khaveh
ztv821

Morten Poulsen
snq880

July 9, 2017

Introduction

The focus of this project is to parallelise the sequential version of the Crank-Nicolson finite difference method. In order to do this, we went through a number of steps to arrive at the optimized CUDA version of the code. In this report, we take you through all the steps we took and the speedup we get at each stage (or otherwise).

Nomenclature

There are a lot of loops in the code, so clear naming of individual loops and code sections are needed. In the following, we will refer to sections of code by name e.g. setPayoff to refer to the contents of the setPayoff function, even after the contents is inlined, or by name from comment, e.g. explicit x for the explicit x loop in rollback. When looking at a section of code, we will refer to individual loops by the loop range variable, e.g. outer refers to the loop that iterates over i in the range $[0, outer[$.

Direction vectors

It is pertinent to discuss about the direction vectors because they are the basis upon which we made our choices. We discuss in reference to the original sequential implementation.

Cross-iteration dependencies

In this section, we have the same pattern of cross-iteration dependencies between the outer loop and the inner loops of the functions (setPayoff, updateParams and rollback). As shown in Listing 1, it can be seen that we write into the same locations of globs.myResult in iterations of the outer loop. This same pattern is seen in the other functions. This implies that the direction vector we have with respect to the outer loop is [>]. The pattern is shown below:

```
for( unsigned k = 0; k < outer; ++ k ) {
  ...
  A[k] =  A[k+1];
```

Set payoff

```
REAL strike;
PrivGlobs    globs(numX, numY, numT);

for( unsigned k = 0; k < outer; ++ k ) {  //loop1
    strike = 0.001*k;
    ...
    // setpayoff
    for(unsigned i=0;i<globs.myX.size();++i)  //loop2
    {
        REAL payoff = max(globs.myX[i]-strike, (REAL)0.0);
        for(unsigned j=0;j<globs.myY.size();++j)  // loop3
            globs.myResult[i][j] = payoff;  //S1
            //Inlined- globs.myResult[i][j] = max(globs.myX[i]-strike, (REAL)0.0);
    }
}
```

Listing 1: setPayoff

- Cross-iteration dependencies: From Listing 1, we observe that there is a WAW dependency with respect to loop1, this is because in each iteration of the outer loop, we write to the same memory location of globs.myResult in S1.We eliminate this dependency by an array expansion.
  With respect to loop2 and loop3, there is a RAW false dependency because of variable payoff which can be easily resolved by inlining the variable payoff in loop3, as indicated in Listing 3.
- Intra-iteration dependencies: There are no dependencies within this loop nest.

Update parameters

```
for( unsigned i = 0; i < outer; ++ i ) { //loop1
    ...

    // updateParams
    for(int g = numT-2;g>=0;--g){ //loop2
                for(unsigned i1=0;i1<numX;++i1) //loop3
                    for(unsigned j=0;j<numY;++j) { // loop4
                        globs.myVarX[i1][j] = exp(2.0*(  beta*log(globs_inv.myX[i1])
                                                      + globs_inv.myY[j]
                                                      - 0.5*nu*nu*globs_inv.myTimeline[g] )
                                                );//S1
                        globs.myVarY[i1][j] = exp(2.0*(  alpha*log(globs_inv.myX[i1])
                                                      + globs_inv.myY[j]
                                                      - 0.5*nu*nu*globs_inv.myTimeline[g] )
                                                ); // nu*nu // S2
                    }

            ...

    }
```

Listing 2: updateParams

- Cross-iteration dependencies: From Listing 2, we observe that there is a WAW dependency with respect to loop1, this is because in each iteration of the outer loop, we write to the same memory locations of globs.myVarX and globs.myVarY. We eliminate this dependency also by an array expansion.With respect to the other inner loops, we have no dependencies as we only read from the variables.
- Intra-iteration dependencies: There are no dependencies within this loop nest.

Rollback

```
   for( unsigned i = 0; i < outer; ++ i ) { //loop1

      for(unsigned i=0;i<globs.myX.size();++i)
     {
      REAL payoff = max(globs.myX[i]-strike, (REAL)0.0);
      for(unsigned j=0;j<globs.myY.size();++j)
        globs.myResult[i][j] = payoff;
     }
      ...
      /* updateParams writes into globs.myVarX and globs.myVarY */
      ...

      /* rollback STARTS */
      ...
      // implicit y
      for(i=0;i<numX;i++) {
          for(j=0;j<numY;j++) {  // here a, b, c should have size [numY]
              a[j] =     - 0.5*(0.5*globs.myVarY[i][j]*globs.myDyy[j][0]);
              b[j] = dtInv - 0.5*(0.5*globs.myVarY[i][j]*globs.myDyy[j][1]);
              c[j] =     - 0.5*(0.5*globs.myVarY[i][j]*globs.myDyy[j][2]);
          }
          ...
          tridag(a,b,c,y,numY,globs.myResult[out][i],yy);
      }
      ...
      /* rollback ENDS */
  }
```

Listing 3: rollback

- Cross-iteration dependencies: This is a similar case to the functions above, from Listing 3, we observe that there is a WAW dependency with respect to loop1, this is because in each iteration of the outer loop, we write to the same memory locations of u,v,a,b,c and y. We eliminate this dependency by array expansions.
- Intra-iteration dependencies: From Listing 3, we can see that implicit y nest loop in rollback() function reads globs.myVarY written which is written by updateParams() in same iteration. This applies for globs.myVarX in the call for implicit x too. It can also be observed that tridag

in implicit y loop nest, reads globs.myResult, which is written by setPayoff prior to call to rollback within the same iteration. Therefore, it is clear to see that there are RAW dependencies for globs.myVarX, globs.myVarY and globs.myResult with respect to loop1 in this case.

Parallelizing the CPU implementation

Step 1 - Privatization of Globs

In the file **ProjCoreOrig.cpp**, the first step we took was to move the declaration of strike and PrivGlobs. We did this because there were cross iteration WAW dependencies in the outer loop which could be eliminated by privatisation of the variables, meaning that every iteration of the outer loop will have their copy of these variables hereby removing the dependencies.

This is safe because the variables are overwritten before use by different iterations of the outer loop, making these dependencies false dependencies. As declaring the variables outside of the loop avoids allocating and deallocating memory in each iteration, this privatization is likely to increase the time usage.

Step 1.1 - OpenMP parallelization

The next step we took was to parallelise the outermost loop of runOrigCPU; it suffices to argue that no dependencies carried between iterations of the outermost loop exist i.e. the reads are covered by writes. In this case, this reduces to show that all locations read from globs and strike in one iteration are covered by writes in the same iteration. This is easily verified as initGrid(), initOperator() and setPayoff() completely overwrite all arrays of globs every time the value() function is invoked.It is shown in Listing 4.

```
#pragma omp parallel for default(shared)schedule(static)
for( unsigned i = 0; i < outer; ++ i ) {
    REAL strike;
    PrivGlobs globs(numX, numY, numT);
    strike = 0.001*i;
    res[i] = value( globs, s0, strike, t,alpha, nu, beta,numX, numY,numT );
}
```

Listing 4: Outer loop with OpenMP pragma

Table 1: Speedup for step 1

Time in micro seconds	Small dataset	speed up	Medium dataset	speed up	Large dataset	speedup
Original	2012534	1.0	4359110	1.0	197702811	1.0
Move Declaration	2131597	0.9	4286231	1.0	209335533	0.9
OpenMP	154496	13	239643	18	9891498	20

From Table 1, it can be seen that the speedup achieved for small dataset when using OpenMP is $\times 13$ and $\times 20$ for the large dataset. The CPU has 32 threads which is the theoretical bound on the maximum speedup that can be obtained. In terms of programming effort OpenMP is easy and results in a good speedup.

Step 2

Step 2.1 - Inlining small functions

To reason properly about loop-level parallelism and apply suitable transformations to exploit it, we made all loop nests explicit by inlining functions that are called by runOrigCPU(). All function bodies were inlined except for tridag() and rollback().

Step 2.2 - Move calls to Inits

We observed that the function calls to initGrid and initOperator were invariant of the outer loop so we moved them out of the value function into the runOrigCPU above the pragma. In order for this to compile correctly, we needed to move the declaration of PrivGlobs above the pragma. This step reduces the amount of computations significantly. The code is shown in Listing 5. Table 2 shows the running time after Moving the PrivGlobs declaration (step 2.2) and Inlining steps (step 2.1).After inlining the functions, the outer loop was no longer parallel, consequently, OpenMP was not used for that step.

Table 2: Speedup for step 2

Time in micro seconds	Small dataset	speed up	Medium dataset	speed up	Large dataset	speedup
Original	2012534	1.0	4359110	1.0	197702811	1.0
OpenMP	154496	13	239643	18	9891498	20
Inline	2011414	1.0	4296713	1.0	191477686	1.0

Step 3

Step 3.1 - Globs constructors

We observed that some variables in PrivGlobs do not change during the outer loop - they are invariant. Other parts do change in each iteration of the outer loop. Thus, it makes sense to split PrivGlobs into two structures, grouping the invariant parts into PrivGlobsInv, and the non-constant parts into ExpGlobs.

Step 3.2 - Array Expansion of Globs

Given that the non-constant parts of PrivGlobs are written and read in each iteration of the outer loop, we have a cross iteration WAW dependency. This type of dependency can be solved by array expansion of the non-constant variables grouped in ExpGlobs. This transformation does not change the semantics of the code, because we only allocate additional memory, in order for each iteration of the outer loop to use its own instance of the variables. It does require an additional indexing variable, though. As can be seen on Table 4, the array expansion decreased the speedup. This is

Table 3: speedup for step 3.1

Time in micro seconds	Small dataset	speed up	Medium dataset	speed up	Large dataset	speedup
Original	2012534	1.0	4359110	1.0	197702811	1.0
Move Inits and Declaration	150189	13.4	229331	19	9873389	20
Inline	2011414	1.0	4296713	1.0	191477686	1.0
Add constructor	2020362	1.0	4344438	1.0	192442301	1.0

likely because we allocate a lot more memory, which is a sequential operation, Amdahl's Law tells the parallel performance is decreased as sequential operation increases. The time used could be reduced by expanding over the number of threads rather than the size of outer, but as our goal is an efficient CUDA implementation this is ignored.

Table 4: speedup for step 3.2

Time in micro seconds	Small dataset	speed up	Medium dataset	speed up	Large dataset	speedup
Original	2012534	1.0	4359110	1.0	197702811	1.0
Add constructor	2020362	1.0	4344438	1.0	192442301	1.0
Array Expansion	1083780	1.9	2279815	1.9	110289223	1.8
OpenMP	154496	13	239643	18	9891498	20
Array Expansion with OpenMP	220442	9.1	501239	8.7	14788748	13.4

Step 4 - Loop distribution

Next we perform a loop distribution, where the outer loop is distributed over the inner loops. This is safe because we have already done an array expansion so there are no longer output dependencies and it is always safe to distribute over a parallel loop.

Step 4.1 - Loop interchange

Loop interchanging is done when we want the sequential loops to be the outermost one, so that we can parallelize the ones inside it with a kernel that does not need to do anything sequential. It should noted that when we did the loop interchange, the outer loop was parallel, therefore, we used OpenMP in this step. This is done on the timestep loop and the outer loop after distributing the outer loop around the timestep loop. The two loops are as follows:

```
for(unsigned o = 0; o < outer; ++o)
    for(int g = numT-2;g>=0;--g) // Sequential
```

We interchange them to become:

```
for(int g = numT-2;g>=0;--g) // Sequential
    for(unsigned o = 0; o < outer; ++o)
```

This can be done since there are no cross iteration dependencies in the outer loop (i.e. it is parallel) and we are always allowed to interchange a parallel loop inwards.We also interchange loops to support coalesced memory access. Consider the following example:

```
for(j=0;j<numY;j++)
    for(i=0;i<numX;i++)
        v[i][j] = ...
```

Currently, the vector of v is being accesses in a sub-optimal way that does not minimize cache misses (by the principle of locality of reference), so interchanging the loops will allow for less cache misses and faster execution. In general, we want to match the order of loops to the indexing, i.e. the outermost loop is the index to the outermost dimension, the innermost loop is the index into the innermost dimension, and so forth. So the optimal version of the above loop would be:

```
for(j=0;j<numY;j++)
    for(i=0;i<numX;i++)
        v[j][i] = ...
```

Table 5: speedup for step 4

Time in micro seconds	Small dataset	speed up	Medium dataset	speed up	Large dataset	speedup
Original	2012534	1.0	4359110	1.0	197702811	1.0
Array Expansion with OpenMP	220442	9.1	501239	8.7	14788748	13.4
Loop distribution without OpenMP	2032934	0.99	4357173	1.0	192106556	1.0
Loop interchange on updateParams	211442	9.5	290221	15	13513266	14.6

Step 5 - Flattening data structure

The code is at this point using vectors, which we would want to rewrite them as flat arrays to make the transition to CUDA kernels easier. When flattening out an array, you have to allocate it differently and index into it differently. Consider an example. The array expanded myResult vector inside the ExpGlobs structure is:

`vector<vector<vector<REAL> > > myResult`

This has dimensions [outer][numX][numY]. When allocating the flattended memory outer*numX*numY*sizeof(REAL) bytes are needed. When indexing, you would have to do it like this:

```
myResult[out][i][j] -> myResult[out*numX*numY + i*numY + j]
```

Basically you multiply each index with the size of the array that are to be simulated to be inside it, and then add them together. This was done on all vectors in the program, meaning the vectors inside the ExpGlobs and PrivGlobs structures, and also the vectors u , v , a , b , c , y and yy . Compared

Table 6: speedup for step 5

Time in micro seconds	Small dataset	speed up	Medium dataset	speed up	Large dataset	speedup
Original	2012534	1.0	4359110	1.0	197702811	1.0
OpenMP	154496	13	239643	18	9891498	20
Change data structures	333033	6.0	435034	10	16754490	12

to the previous step, we observe a significant slowdown from this modification. This could be caused by the compiler being able to recognize and optimize access patterns when using vectors, but not for the flattened arrays.

CUDA implementation

Naive CUDA Kernels

The Naive CUDA kernels were directly derived from the already parallelized sequential version of the code. They are placed in CudaKernels.cu.h, and also includes CUDA versions of the initGrid and initOperator functions to avoid copying them to the GPU. The naive CUDA kernels have some simple optimizations to reduce access to global memory, e.g. when filling u in the explicit x loop, we write the temporary results into a local variable and write the result to global memory. The speedup for the naive CUDA implementation is shown in Table 7.

In order to wrap the calls to cudaMalloc and cudaFree, we implemented CUDA versions of PrivGlobsInv and ExpGlobs. These are placed in ProjHelperFunCuda.h.

By commenting out parts of the code, it is seen that without the calls to the tridag kernel, execution takes some 4 seconds, so the major part of the execution time is spent in the tridag kernel. So, any optimization efforts should focus on that kernel. Looking at the CPU implementation, it can be seen that in each iteration of the forward and backward loops, the value saved into u and uu in the previous iteration are used. These can be saved into a local variable, thereby reducing the number of global memory accesses by $3n$ for each call to tridag where n is the length of the row. The result is shown in Table 7. As can be seen, the time usage is reduced by ~ 2.5 seconds, underlining how costly global memory accesses are.

The improvement in tridag suggests that the memory handling of tridag should be improved. There are a number of general approches to improving the memory handling of a kernel:

- Use local variables: Cache repreated global memory accesses into local variables and use those. This is what we have done.

- Coalescing: Modify memory access patterns, so all threads in a block access memory areas that are close to each other in each instruction.
- Shared memory: Copy data from global to shared memory, use it, and copy output from shared to global when done.

The following sections will look into the optimizations of tridag that we have attempted.

The rollback function uses a lot of scratchpad memory that is allocated and deallocated in each iteration. This memory allocation has been moved outside of the numT loop, so the memory areas are reused. This also helps us ensure that we do not leak GPU memory as CUDA fails silently - kernels just return without doing anything. As can be seen in Table 7, this significantly improves performance for the smaller datasets, but has little effect on the large dataset.

Table 7: Speedup for Naive CUDA implementation. The performance data for CUDA has not been cleaned of the start up time for CUDA, which is on the order of $150,000\mu s$

Time in micro seconds	Small dataset	speed up	Medium dataset	speed up	Large dataset	speedup
Original	2012534	1.0	4359110	1.0	197702811	1.0
OpenMP	154496	13	239643	18	9891498	20
Naive CUDA	479960	4	563752	8	15805424	13
Fewer tridag global accesses	445929	5	540154	8	13338663	15
Scratchpad memory	349104	6	432885	10	13099319	15

Shared memory in Tridag

We need to read the four input arrays (a, b, c, r) and write to u, suggesting that at least $5n$ global memory accesses are needed by each tridag kernel where n is the length of each row. With the optimizations to global memory above, tridag has $10n$ accesses to global memory. Shared memory is a lot faster to access, so if the needed data is copied from global memory to shared memory, it can be accessed much faster and written only once. However, this also requires that each thread has access to $8 \cdot 6n = 48n$ bytes of shared memory as we need to hold 6 arrays of doubles (a, b, c, r, u and uu).

This can feasibly be achieved by letting the threads of each block cooperate on copying the input arrays into shared memory. Each block of threads working on tridag can cooperate on copying data coalesced from global memory to shared memory. There is an important trade-off here, as each block has a limited amount of shared memory. As the length of each row increases, the memory needed to keep it also increases. It follows that the number of rows that can be in shared memory decreases. But tridag is sequential, so efficient execution requires that all threads in the block are working. So, if the number of rows falls below the number of threads available, a lot of threads are going to be idle.

Our attempts to find a sweet-spot loading c into shared memory did not pay off: In all instances we ended up with a time usage that was worse that with the naive CUDA implementation. It also caused a validation error for the medium dataset, so there is likely an error somewhere. The speedup is shown on Table 8.

This suggests that in order to get any improvement from using shared memory, tridag must be parallelized first.

Table 8: Speedup for Shared Memory CUDA implementation. The performance data for CUDA has not been cleaned of the start up time for CUDA, which is on the order of $150,000\mu s$.

Time in micro seconds	Small dataset	speed up	Medium dataset	speed up	Large dataset	speedup
Original	2012534	1.0	4359110	1.0	197702811	1.0
OpenMP	154496	13	239643	18	9891498	20
Naive CUDA	479960	4	563752	8	15805424	13
Shared Memory CUDA	1364820	1.5	failure	-	31166295	6

The small dataset sticks out. We have observed that after Friday the 4th, the time usage of our kernels on the GPU machine increased from on the order of $400,000\mu s$ to the order of $1,300,000\mu s$. On another computer with a GFX 780 Ti, the time usage was unchanged.

Coalescing Tridag

Coalescing memory accesses is an optimization on the GPU similar to locality of reference on the CPU. The intent is to optimize global memory access, so time spent waiting on cache misses is as small as possible. In CPU, this can be achieved by letting the threads in each block access global memory areas that are close to each other. On the GPU each thread can access memory in parallel in the same instruction. To achieve good cache coherency these memory accesses should also be close to each other. From the point of view of the individual thread this is equivalent with accessing in column order.

This is very different from the naive tridag implementation where each thread is iterating over a number of rows in a number of arrays. This means that each thread in each block is accessing wildly different memory areas.

To implement this idea in the optimal tridag kernel, we just had to make sure that the innermost index is the thread index which is possible by accessing variables in a transposed fashion, as shown below:

```
REAL a = arr[n + threadIdx.x]
```

This results in a pattern where e.g. 32 threads access 32 concurrent memory locations in arr.

Coalescing other kernels

Aside from coalescing the tridag kernel, we also coalesced the memory access of some of the other kernels. In the explicit x CUDA kernel, we coalesced the writes to u by accessing it in a transposed manner u. We used the output from explicit x in explicit y by accessing it in a transposed manner as well.

Table 9: Overall results. The last line was run with a CUDA dummy kernel before timing started.

Time in micro seconds	Small dataset	speed up	Medium dataset	speed up	Large dataset	speedup
Original	2012534	1.0	4359110	1.0	197702811	1.0
OpenMP	154496	13	239643	18	9891498	20
Naive CUDA	479960	4	563752	8	15805424	13
Coalesced CUDA	272975	7	309446	14	3772178	52
Coalesced CUDA w dummy kernel	253831	8	316675	14	3765984	52

Results and Discussion

Compared to the sequential implementation, we have achieved a significant improvement with both the OpenMP ($\times 20$ speedup) and the CUDA ($\times 52$ speedup) implementations for the large datasets. The CUDA speedup for the small and medium datasets are significantly smaller, and not as good as the OpenMP speedup. For the small dataset this is likely due to tridag not being parallel, which could cause GPU threads to be idle. The medium dataset has dimensions that are not divisible by 32, making them sub-optimal for CUDA.

The CPU on the GPU machines is a powerful Intel Xeon CPU E5-2650 v2 @ 2.60GHz with 8 cores supporting 32 threads, and the GPU is an nVidia GFX 780 Ti.

```
void   run_OrigCPU(
                const unsigned int&   outer,
                const unsigned int&   numX,
                const unsigned int&   numY,
                const unsigned int&   numT,
                const REAL&           s0,
                const REAL&           t,
                const REAL&           alpha,
                const REAL&           nu,
                const REAL&           beta,
                      REAL*           res   // [outer] RESULT
) {
    PrivGlobs    globs(numX, numY, numT);
    initGrid(s0,alpha,nu,t, numX, numY, numT, globs);
    initOperator(globs.myX,globs.myDxx);
    initOperator(globs.myY,globs.myDyy);

    #pragma omp parallel for default(shared) schedule(static)
    for( unsigned i = 0; i < outer; ++ i ) {
        REAL strike;

        strike = 0.001*i;
        res[i] = value( globs, s0, strike, t,
                        alpha, nu,    beta,
                        numX,  numY,  numT );
    }
}
```

Listing 5: Move declaration and inits