Local-Volatility Calibration

University of Copenhagen

October 30, 2020

Justinas Antanavičius rqh977@alumni.ku.dk

Francesco Done' qwg586@alumni.ku.dk

Valdemar Erk shm302@alumni.ku.dk

Exchange Student

Abstract

Our aim with this paper is to explain how it was possible to parallelize the code [1] that implements a simplified version of volatility calibration, which uses Crank-Nicolson finite difference method [2]. In particular, in this document, there are two sections, in which are defined both parallelizations: CPU via *OpenMP* and GPU via *CUDA*. In each of them, there are described all the operations done step-by-step and also the speedups obtained with the parallel versions of the program, compared to the sequential one.

1 CPU parallelization

Parallelize the code on CPU via *OpenMP* is trivial, since it is sufficient to privatize the variables `strike` and `globs`, and then apply the `#pragma` directive as follows:

```
1  //code\OrigImpl\ProjCoreOrig.cpp
2
3  void run_OrigCPU(/*...*/) {
4    #pragma omp parallel for
5    default(shared) schedule(static)
6    if(outer>8)
7      for(unsigned i=0; i<outer; ++i) {
8        REAL strike;
9        PrivGlobs globs(numX, numY, numT);
10       strike = 0.001*i;
11       res[i] = value( globs, s0, strike,
             t, alpha, nu, beta, numX,
             numY, numT );
12     }
13 }
```

It is safe to apply because the iteration i reads (in line 11) what was written in the same iteration (in lines 8 and 9), so the variables can be privatized.

1.1 Transformations

First of all, it was tried to directly apply parallelization techniques to each function in the CPU implementations with OpenMP pragmas. This leads to some speedup, but the overhead of creating threads and splitting the work between them was enormous. This meant that some parallelization was not efficient enough to give a speedup. Then, starting from the beginning, transformations were made to get the code more suitable for re-implementation on GPU. The first transformation we did was in-lining nearly all function called in the primary function and continued to do that such that we had a sizeable continuous function. We then did array expansion as described later in the document. Furthermore, at this point, we then began to implement the GPU kernels slowly.

1.2 Speedup

The execution time decreased significantly implementing the CPU parallelization, as is possible to see in the following table:

Dataset	**W/ (μS)**	**W/O (μS)**
Small	234362	2429592
Medium	513036	5240587
Large	16026744	232179276

Table 1: Execution time with different datasets, with (W/) and without (W/O) CPU parallelization.

Therefore, the speedups obtained with the above code implemented are resumed in the following table:

Dataset	**Speedup**
Small	937%
Medium	921%
Large	**1349%**

Table 2: Speedup obtained parallelizing CPU via OpenMP with different datasets.

2 GPU parallelization

Parallelize the code on GPU via *CUDA* requires a case study. From a high-level point of view, the memory needs to be allocated where the kernel is started, so each loop iteration i has exclusive access to a memory location. Therefore, it writes to and reads from, row i of expanded array A, in this way it is possible to perform array expansion [2.1]. Then, in order to improve the degree of parallelism, loop distribution is achieved [2.2] while it is necessary to manage loop interchange [2.3] and matrix transposition [2.4] because they help to achieve coalesced memory access. Moving objects to shared memory and inlining functions help in reducing the number of accesses to it [2.5]. The speedup gained from the implementation of all of these techniques is described at the end of this chapter [2.6], comparing it with the sequential version of the GPU handed-in code.

2.1 Array Expansion

When running code in a parallel fashion, each parallel thread must not work on the same memory as another thread. Access like that could lead to race conditions and incorrect results. A technique to alleviate this is to expand the memory such that each thread only touches its part of the memory. So you go from having a D-dimensional array to a $D+1$-dimensional array with as many rows in the new dimension as you have threads.

Thread id	**Memory addresses**			
Thread 0..N	\$0	\$1	$\cdots$	\$M

Table 3: 1-dimensional array with multiple threads accessing the same memory locations $\$i$

In the above example we can see that all threads from 0 to N access the same memory specifically \$1 to \$M. This will almost certainly cause race conditions to happen, especially if the memory is used after it is modified, as it may then be in the state needed by another thread.

Thread id	**Memory addresses**			
Thread 0	\$[0,0]	\$[0,1]	$\cdots$	\$[0,M]
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Thread N	\$[N,0]	\$[N,1]	$\cdots$	\$[N,M]

Table 4: 2-dimensional array with multiple threads memory exclusively

In the above example, we can see that thread 0 will only access memory in the first row of the table, and so on until thread N which is only accessing row N. In the table we use a shorthand for memory location this means that $\$[n, m] = \$[n * M + m]$. This, of course, cause a massive increase in memory usage. In essence, multiplying the memory with the amount of parallelism, but often this is a fair trade-off for the speed increase. In our case with the Large input, this leads to around a 128 times increase in the memory needed. The original CPU implementation used C++ vectors, having vectors of vectors is not possible on GPUs as you need a flat area of memory and cannot have indirections. We first converted it to C arrays of arrays, which again have indirection and therefore are not usable on GPUs. At this step, we did the array expansion as it was easier to index into it as this stage. We then flattened the arrays, so if we had an array $arr[X][Y][Z]$ we would change it to used `malloc(3)` and have the size $X * Y * Z$. To index into a flattened array, we had to write a small formula to index into the array such that the position $arr[x][y][z]$ would be turned into $flat\_arr[x * Y * Z + y * Z + z]$. In some of our cases, we needed to access a sub-array, this is still possible because arrays in C is just pointers and does not contain any size information so we could point to the first element of the array and it would work. Here below, there is a practical example of array expansion:

```
float A[N];
for(int i=0;i<M;i++){
    for(int j=0;j<N;j++){
        A[j] = ...
    }
    for(int j=0;j<N;j++){
        ... = A[j]
    }
}
```

In the code above the outermost loop is sequential since each iteration i reads and writes from and to all the indices of A. The transformed code below demonstrates how to make the outermost loop parallel because now the iteration i reads and writes from and to the row i of expanded array A:

```
float A[M, N];
for(int i=0;i<M;i++){
    for(int j=0;j<N;j++){
        A[i,j] = ...
    }
    for(int j=0;j<N;j++){
        ... = A[i,j]
    }
```

```
}
```

In order to parallelize most of the code, we did array expansion for almost all the arrays used in the function `run_OrigCPU`. Most of the arrays needed to be expanded by one - outer - dimension, e.g. `dtInv, myVarY`, etc., while others by more than one dimension, e.g. `a, b, c`, etc.

2.2 Loop Distribution

With loop distribution, a loop, which has independent parts, is split up into multiple loops such that it more effectively can be run on parallel hardware such as GPUs. The gist of the transformation can be shown in the following way so if we have a loop such as the following:

```
len = len(X) = len(Y)
for i in len do
   X[i] = f(i)
   Y[i] = g(i)
end
```

In the above loop, X and Y are completely independent of each other, this means that we can then split it up into two loops in the following way:

```
len = len(X) = len(Y)
for i in len do
   X[i] = f(i)
end

for i in len do
   Y[i] = g(i)
end
```

A place where we did loop distribution was moving a function call out of a nested loop such that we had a perfectly nested loop which could be parallelized more effectively. The code before the transformations was the following

```
// implicit x
for( unsigned k = 0; k < outer; ++ k ) {
  for(unsigned j=0;j<numY;j++) {
    for(unsigned i=0;i<numX;i++) {
      a[k][i] = ...;
      b[k][i] = ...;
      c[k][i] = ...;
    }
    tridagPar(a[k],b[k],c[k],...);
  }
}
```

We then expanded the three arrays `a`,`b` and `c` with one dimension, this means that we could put each run of each dimension into its own ”layer” of our three dimensionel matrix. We then changed the code to the following:

```
// implicit x
for( unsigned k = 0; k < outer; ++ k ) {
   for(unsigned j=0;j<numY;j++) {
      for(unsigned i=0;i<numX;i++) {
         a[k][j][i] = ...;
         b[k][j][i] = ...;
         c[k][j][i] = ...;
      }
   }
}

for( unsigned k = 0; k < outer; ++ k ) {
  for(unsigned j=0;j<numY;j++) {
    tridagPar(a[k][j],b[k][j],c[k][j],...);
  }
}
```

This meant that we could split the resulting CUDA code into two kernels to execute it in a better performing way.

2.3 Loop Interchange

Loop interchange is possible when you have nested loops in which permuting the direction matrix, in the same way, does not result in a $>$ direction as the leftmost non-$=$ direction in a row (Legality of Loop Interchange Theorem [3]). This help in managing coalesced memory access, for example, the code below

```
for (i=2; i<N; ++i){
   for (j=2; j<N; ++j){
      A[j,i]=A[j-1,i]//... S1
      B[j,i]=B[j-1,i-1]//... S2
   }
}
```

can be permuted as

```
for (j=2; j<N; ++j){
   for (i=2; i<N; ++i){
      A[j,i]=A[j-1,i]//... S1
      B[j,i]=B[j-1,i-1]//... S2
   }
}
```

because the direction matrix was

$$\begin{cases} [=,<] & \text{S1} \rightarrow \text{S1} \\ [<,<] & \text{S2} \rightarrow \text{S2} \end{cases}$$

and now it is

$$\begin{cases} [<,=] & \text{S1} \rightarrow \text{S1} \\ [<,<] & \text{S2} \rightarrow \text{S2} \end{cases}$$

This means that the innermost loop is now parallel since the outer one is carrying all the dependencies, also since the innermost loop leads the index i, it optimizes the spatial locality for the access to `A[j,i]` and `B[j,i]`.

In our code we faced an issue when sequential loop was in between parallel loops:

```
for(unsigned k=0; k<outer;++ k) { // par
   for(int g = globs.sizeT-2;g>=0;--g)
       { // seq
      for(unsigned
          i=0;i<globs.sizeX;++i) { //par
         ...
      }
   }
}
```

To increase the degree of parallelism, we interchanged sequential loop g with parallel loop k. This is safe because loop k is parallel; thus, it can be interchanged inwards. Now the outermost loop is sequential while all inner loops are parallel.

```
for(int g=globs.sizeT-2;g>=0;--g) { //
    seq
   for(unsigned k=0; k<outer;++k) {//par
      for(unsigned
          i=0;i<globs.sizeX;++i) { //
          par
         ...
      }
   }
}
```

We did loop intechange not only to increase the degree of parallelism, but also to get coalesced memory access. For example, we interchanged loops i and j in function explicit y to insure coalesced memory access of array v. Function before transformation:

```
// explicit Y
for( unsigned k = 0; k < outer; ++ k ) {
  for(unsigned j=0;j<numY;j++) {
    for(unsigned i=0;i<numX;i++) {
      v[k][i][j] = 0.0;
      ...
    }
 }
}
```

Function after interchanging loops i and j:

```
// explicit Y
for( unsigned k = 0; k < outer; ++ k ) {
  for(unsigned j=0;j<numY;j++) {
    for(unsigned i=0;i<numX;i++) {
      v[k][i][j] = 0.0;
      ...
    }
 }
}
```

2.4 Matrix Transposition

Matrix transposition can be used when you have two or more dimensions in your data. It is used to make memory access coalesced. It is also used for mathematical operations although we do not use it for such. We have implemented a tiled transposition on a 3-dimensional array, such that if we have an array $arr[z][x][y]$ we can transpose it into an array $arr^T[z][y][x]$ so that we transpose each layer as a 2-dimensional matrix. This gives us better-coalesced memory access when iterating over the inner matrix on each layer. Our specific implementations use a tiled transpose such that we transpose in blocks as that is more effective on GPUs.

Example when matrix transposition helps to achieve coalesced memory access:

```
for(int k = 0; k < num_k; k++) {
   for(int j=0; j<num_j; j++) {
      for(int i=0; i<num_i; i++) {
         u[k][j][i] = 3 * v[k][i][j];
      }
   }
}
```

In this example, u has coalesced memory access, while v has uncoalesced memory access. Interchanging loops j and i would not help in this case because v would have coalesced memory access while u would have uncoalesced memory access. To make sure that both, u and v, have coalesced memory access, we have to transpose two innermost dimensions of the array v:

```
// v[num_k][num_i][num_j]
// v_T[num_k][num_j][num_i]
v_T = transpose_matrix(v);
for(int k = 0; k < num_k; k++) {
  for(int j=0; j<num_j; j++) {
    for(int i=0; i<num_i; i++) {
      u[k][j][i] = 3 * v_T[k][j][i];
    }
  }
}
v = transpose_matrix(v_T);
```

Moreover, it is safe to apply this transformation because we transpose array back to its original form i.e.
$v = transpose\_matrix(transpose\_matrix(v))$.

We were planning to apply matrix transposition for innermost two dimensions of array u to insure coalesced memory access. Example of function from our code without coalesced memory access to array u but with coalesced memory access to array $globs.myResult$:

```
// explicit x
for( unsigned k = 0; k < outer; ++ k ) {
  for(unsigned i=0;i<numX;i++) {
    for(unsigned j=0;j<numY;j++) {
      u[k][j][i] =
          dtInv[k]*globs.myResult[k][i][j];
      ...
```

```
    }
  }
}
```

The same function with matrix transposition and coalesced memory access to both u and $globs.myResult$:

```
// explicit x
u_T = transpose(u);
for( unsigned k = 0; k < outer; ++ k ) {
  for(unsigned i=0;i<numX;i++) {
    for(unsigned j=0;j<numY;j++) {
      u_T[k][j][i] =
          dtInv[k]*globs.myResult[k][i][j];
      ...
    }
  }
}
u = transpose(u_T);
```

Unfortunately, the program was not validating with the implemented matrix transposition kernel even though transposing array twice did not change it. Because we could not find the issue, we did not do any matrix transposition in the final code.

2.5 Inlining

Since the code has unnecessary memory accesses, we could optimize them away to gain execution-speed. In our case, this was for example done by inlining: we in-line some calculations which would otherwise have caused memory access in the GPU implementation. For example, we had a variable `strike` which was in our code after array expansion equal to the following code:

```
for( unsigned k = 0; k < outer; ++ k ) {
  strike[k] = 0.001*k;
}
```

which would cause memory access in the kernel `initPayoff` but if we inlined it, it is only a single floating-point multiplication which is much faster. The following is a part of the code in the `initPayoff` kernel before we inlined `strike`, as you can see we are doing a memory access when we get the strike value.

```
if (gidy < outer && gidx < numX) {
 payoff_cuda[gidy*numX+gidx] =
   max(myX[gidy*numX+gidx]-strike[gidy],
       (REAL)0.0);
}
```

This is the same code after we inlined `strike`, as you can see here we just do a single multiplication which is much faster than a memory access.

```
if (gidy < outer && gidx < numX) {
  payoff_cuda[gidy*numX+gidx] =
    max(myX[gidy*numX+gidx]-gidy*0.001,
        (REAL)0.0);
}
```

2.6 Speedup

Since the new code is optimized to work in a Nvidia Graphics Card, its execution time results pretty fast as expected [2.6.1]. The obtained GPU parallel version of the code results much faster compared with the CPU sequential one [2.6.2]. The comparison was performed with the diku server GPU04 [4] that has these specs:

- 1 x Supermicro SYS-7047GR-TPRF [4U/-Tower barebone LGA2011, 2x1620W PSU, 8x3.5" htswp trays]
- 2 x Intel Xeon E5-2650v2 [8-core CPU, 2.6GHz, 20MB cache, 8GT/s QPI]
- 8 x Samsung 16GB [DDR3 (128GB total) 1866MHz Reg. ECC server module]
- 2 x nVidia GeForce GTX 780 Ti [3072MB, 384 bit GDDR5, PCI-E 3.0 16x]
- 1 x Intel S3500 serie [240GB SATA SSD]
- 1 x Seagate Constellation ES.3 [4TB 7200RPM SATA 6Gb/s 128MB cache 3,5" HDD]

2.6.1 Application total runtime

Dataset	Execution time (μS)
Small	313337
Medium	411741
Large	7211866

Table 5: Execution time with different datasets, with GPU parallelization.

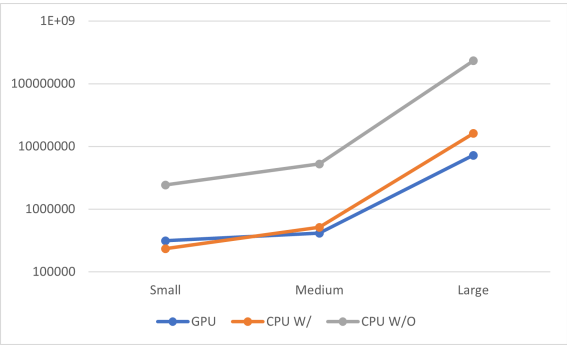

Figure 1: Runtime comparison (μS) between parallel GPU, CPU with optimizations (CPU W/) and CPU without optimizations (CPU W/O)

As it is possible to see in the above chart, the slowest with every database is CPU without optimizations, alternatively, for the small one and the medium one, CPU with optimizations works better that GPU orchestration. Parallel GPU program is the fastest one with a large dataset.

2.6.2 Comparison with the parallel CPU execution

Dataset	**GPU** (μS)	**CPU** (μS)
Small	313337	2429592
Medium	411741	5240587
Large	7211866	232179276

Table 6: Execution time of GPU parallel version in comparison with the sequential CPU execution.

Dataset	**Speedup**
Small	675%
Medium	1173%
Large	**3119%**

Table 7: Speedup obtained parallelizing GPU via CUDA, with respect to sequential CPU, using different datasets.

Conclusions

The implementation of the presented techniques [5] has contributed to making the program running faster, especially with the large dataset, which is 3119% faster than the handed-in one.

References

[1] Handed-in code. code.tar.gz.

[2] Jost Berthold Martin Elsman Fritz Henglein Troels Henriksen Maj-Britt Nordfang Christian Andreetta, Vivien Bégot and Cosmin E. Oancea. Finpar: A parallel financial benchmark. pages 14–15, June 2016.

[3] Cosmin E. Oancea. Pmph lecture notes for the software track. 1:74–75, September 2018.

[4] Gpu04 specifications. link.

[5] Handed-out code. code_optimized.tar.gz.