

# **Programming Massively Parallel Hardware**

## **Exam Group Project**

Arni Asgeirsson   Rasmus Klett Mortensen  
`{lwf986,rzw867}@alumni.ku.dk`

*Lector*

Cosmin E. Oancea

Department of Computer Science, University of Copenhagen, Denmark

## CPU Parallelization

The given CPU sequential code can be optimized with an OpenOMP<sup>1</sup> pragma. In the outmost **outer** loop in **run\_OrigCPU** we can privatize the *strike* and *globs* variables, and add the OpenOMP pragma to CPU parallelize the loop. The code transformation is safe as *strike* and *globs* are not used accross iterations. Furthermore the outer loop is itself parallelisable as there are no cross iteration dependencies. Each loop iteration independently computes a single value of the final result.

Figure 1 shows how the pragma is used.

```
187 #pragma omp parallel for default(shared) schedule(static)
188   for( unsigned i = 0; i < outer; ++ i ) {
189       PrivGlobs   globs(numX, numY, numT);
190       REAL strike = 0.001*i;
191       res[i] = value( globs, s0, strike, t,
192                      alpha, nu,  beta,
193                      numX, numY, numT );
194   }
```

**Figure 1:** Use of openOMP pragma to achieve CPU parallelization.

When timing the OpenOMP version the code averages on  $257131\mu s \approx 0.25s$  when run with the small data set,  $331305\mu s \approx .33s$  with the medium data set and  $13600746\mu s \approx 13.6s$  with the large data set. Hence these shall be our benchmark requirements for our CUDA version.

---

<sup>1</sup><http://openmp.org/>

## Task1

In this section we are going to target efficient parallelization of the **large** dataset. The main strategy here is to parallelize the two outer loops which corresponds to the *outer* and *numX* or *numY* dimensions. Note that each **tridag** call will be executed sequentially inside each kernel.

First we will discuss the many code transformation we have performed next we will look into the individual kernels and discuss why these are parallelisable.

The handin for Task1 is contained in */ProjectPMPH/*. The subfolder *OrigImpl/* contains the original implementation with the openOMP pragma. *CudaPrepareImpl/* contains all our code transformations before writing CUDA kernels. Some code transformations are very specific to the individual implemented kernels and are hence not included in these files. The *CudaImpl/* subfolder is extended on top of the code found in *CudaPrepareImpl* and implements CUDA kernels. Each subfolder contains a **Makefile** which can be used to execute against the given small, medium and large data files.

After applying all of our code transformations and implemented CUDA kernels we are able to achieve an average running time of  $88667\mu s \approx 0.09s$  for the small data set,  $137371\mu s \approx 0.14s$  for the medium data set and  $1225374\mu s \approx 1.2s$  for the large data set. This is a speed up of  $\times 2.9$ ,  $\times 2.4$  &  $\times 11.1$ , respectively. Yet, still many optimizations are still applicable.

## Hoisting of constant values

As with the openMP version we privatize the *strike* variable. Same argumentation as before applies.

Initializing *globs* allocates the required memory for the variables *myX*, *myDxx*, *myY*, *myDyy*, *myTimeline*, *myVarX* & *myVarY*. Since no cross iteration dependencies exist we can reuse the allocated memory space inbetween each iteration and hoist *globs* out of the loops. Note that this is a reversal of a privatization, and could obscure opportunities for parallelization, however as *myTimeline*, *myXindex*, *myX*, *myYindex*, *myY*, *Dxx* & *Dyy* are all constant accross all iterations and therefore these only need to be computed once for all iterations. Hence we hoist the **initGrid** and **initOperator** out of the loop as we can see in Figure 2.

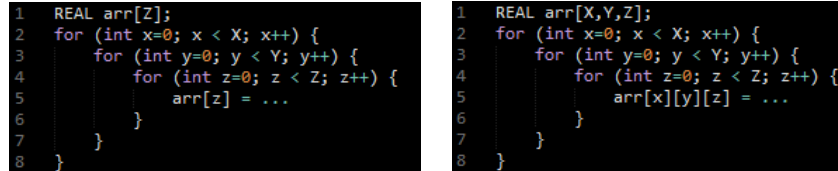
```
186     PrivGlobs    globs(numX, numY, numT);
187     initGrid(s0,alpha,nu,t, numX, numY, numT, globs);
188     initOperator(globs.myX,globs.myDxx);
189     initOperator(globs.myY,globs.myDyy);
190
191     for( unsigned i = 0; i < outer; ++ i ) {
192         REAL strike = 0.001*i;
193         res[i] = value( globs, s0, strike, t,
194                       alpha, nu, beta,
195                       numX, numY, numT );
196     }
```

Figure 2: *initGrid* and *initOperator* hoisted out of the loop nest.

It is clear that this is a safe code transformation, as these variables are never written to after they have been initialized.

## Array Expansion

In a CUDA kernel it is preferable that all memory is allocated before starting it. That means that if we have a privatized variable or array in a loop it is preferable that they are expanded with the outer dimensions such that we can allocate the memory before executing the parallelized kernel. In Figure 3 we can see a generalized array expansion from 1-dim to 3-dim.



```

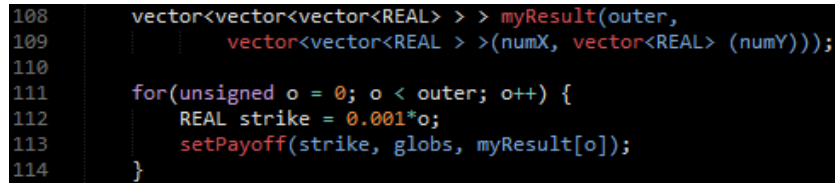
1 REAL arr[Z];
2 for (int x=0; x < X; x++) {
3   for (int y=0; y < Y; y++) {
4     for (int z=0; z < Z; z++) {
5       arr[z] = ...
6     }
7   }
8 }

1 REAL arr[X,Y,Z];
2 for (int x=0; x < X; x++) {
3   for (int y=0; y < Y; y++) {
4     for (int z=0; z < Z; z++) {
5       arr[x][y][z] = ...
6     }
7   }
8 }

```

Figure 3: Pseudo code array expanding an array into its two outer dimensions.

Such a transformation is always safe because each iteration writes and reads to its own row of the array expanded array or variable. This should be given since before the transformation the array or variable was private and hence unaffected by other iterations. We perform exactly such a expansion of the *myResult* array. To do this we also extract it from the *PrivGlobs* struct. *myResult* is now a 3-dim array, and we are able to compute *setPayoff* in a *outer* loop as we can see in Figure 4.



```

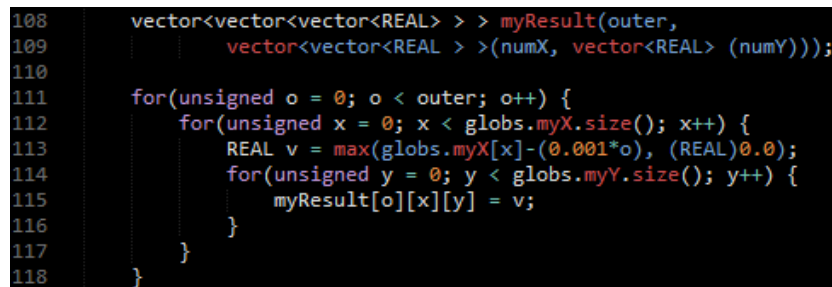
108 vector<vector<vector<REAL> > > myResult(outer,
109   vector<vector<REAL> > (numX, vector<REAL> (numY)));
110
111 for(unsigned o = 0; o < outer; o++) {
112   REAL strike = 0.001*o;
113   setPayoff(strike, globs, myResult[o]);
114 }

```

Figure 4

Array expansion is a key concept we will utilize multiple times.

If we inline *setPayoff* into the loop shown in Figure 4 we can shorten the code and we are left with a loopnest that is fully parallelizable as we can see in Figure 5.



```

108 vector<vector<vector<REAL> > > myResult(outer,
109   vector<vector<REAL> > (numX, vector<REAL> (numY)));
110
111 for(unsigned o = 0; o < outer; o++) {
112   for(unsigned x = 0; x < globs.myX.size(); x++) {
113     REAL v = max(globs.myX[x]-(0.001*o), (REAL)0.0);
114     for(unsigned y = 0; y < globs.myY.size(); y++) {
115       myResult[o][x][y] = v;
116     }
117   }
118 }

```

Figure 5

This loopnest is translated into the *myResultKernel2D* kernel.

It should be clear that there are no loop dependencies between each iteration. We also note that the computed value is constant for each iteration of the inner loop. Hence this computation can be extracted from the inner loop.

## Loop interchange

If we inline the function **value** into the main loop we get what Figure 6 shows.

```
119     for(unsigned o = 0; o < outer; o++) {
120         for(int j = globs.myTimeline.size()-2;j>=0;--j) {
121             updateParams(j,alpha,beta,nu,globs);
122             rollback(j, globs, myResult[o]);
123         }
124     }
125     res[o] = myResult[o][globs.myXindex][globs.myYindex];
126 }
```

Figure 6: The main loop where **value** is inlined.

We can see that we can perform loop distribution on the inner loop and the write to *res*. By doing so, we are now able to loopinterchange the two outer loops calling **updateParams** and **rollback** as we can see in Figure 7.

```
119     for(int j = globs.myTimeline.size()-2;j>=0;--j) {
120         for(unsigned o = 0; o < outer; o++) {
121             updateParams(j,alpha,beta,nu,globs);
122             rollback(j, globs, myResult[o]);
123         }
124     }
125
126     for(unsigned o = 0; o < outer; o++) {
127         res[o] = myResult[o][globs.myXindex][globs.myYindex];
128     }
```

Figure 7: The main loop where a loopinterchange has been performed.

Our loopinterchange is safe since the loop over *outer* is parallel. The final loop, writing to *res* is translated into the **buildResultKernel** kernel in the CUDA version. We note that **updateParams** is constant across all iterations of *outer* and hence it can be extracted out of the loop. If we then inline **updateParams** we get a loopnest that is ready to become a kernel. This code transformation can be seen in Figure 8.

```
119     for(int g = globs.myTimeline.size()-2;g>=0;--g) {
120         REAL nu2t = 0.5*nu*nu*globs.myTimeline[g];
121         for(unsigned x = 0; x < globs.myX.size(); ++x) {
122             for(unsigned y = 0; y < globs.myY.size(); ++y) {
123                 globs.myVarX[x][y] = exp(2.0*( beta*log(globs.myX[x])
124                                     + globs.myY[y]
125                                     - nu2t)
126                                     );
127                 globs.myVarY[x][y] = exp(2.0*( alpha*log(globs.myX[x])
128                                     + globs.myY[y]
129                                     - nu2t)
130                                     ); // nu*nu
131             }
132         }
133
134         for(unsigned o = 0; o < outer; o++)
135             rollback(g, globs, myResult[o]);
136     }
```

Figure 8: The main loop where **updateParams** is inlined.

It makes it more clear that the inlined code is parallelized for a CUDA kernel, as there are no cross iteration dependencies and is translated into the **myVarXYKernel** kernel in our CUDA version. We note that part of the computation is constant across each iteration of the two inner dimensions.

### Loop distribution over rollback

The next code transformation is by moving the *outer* dimensioned loop inside the **rollback** function and loop distribute the dimension across the three main loops inside **rollback**.

We array expand  $u, v, a, b, c, _y$  &  $yy$  to all be of 3 dimensions.

We are now able to identify what will become three kernels inside **rollback** aswell as our main kernels of interest. These are be named **initUAndV2Dim**, **tridag1** & **tridag2**.

## Memory Coalescing

At this point it is interesting to benchmark the performance of the naive CUDA implementation resulting from this. As expected, this implementation is quite slow, taking 25 seconds on large, and the nVidia Profiler confirms that the culprit is uncoalesced memory access.

The runtime is distributed across the kernels like so:

tridag1	8.4s
tridag2	10.0s
initUandV2Dim	7.1s

Since our kernels have the loop over *outer* as their outermost loop, this index is the fastest-varying in the kernels. Therefore the O-index should be used in the innermost indexing into arrays to achieve coalescing. Looking at the dimensions of the arrays we write, this is not the case:

```
a [O, X, Y]
b [O, X, Y]
c [O, X, Y]
yy [O, X/Y, Y/X]
_y [O, Z, Z]
u [O, Y, X]
v [O, X, Y]
myResult [O, X, Y]
```

We therefore need to transpose the arrays, but since these arrays are all written and read from the kernels, we can simply treat the arrays as transposed instead of actually performing transpositions. The indices can therefore be switched so the O-index is the innermost index:

```
a [X, Y, O]
b [X, Y, O]
c [X, Y, O]
yy [X/Y, Y/X, O]
_y [Z, Z, O]
u [Y, X, O]
v [X, Y, O]
myResult [O, X, Y]
```

This provides a dramatic speedup, netting a runtime on large of 1.81 seconds.

tridag1	8.4s
tridag2	10.0s
initUandV2Dim	7.1s



### Parallelization of tridag

It is worth looking into the given function **tridag** and why it is not immediately parallelizable. Inside **tridag** we have a couple of loops which each performs cross iteration dependencies.

```
45     for(i=1; i<n; i++) {  
46         beta = a[i] / uu[i-1];  
47  
48         uu[i] = b[i] - beta*c[i-1];  
49         u[i] = r[i] - beta*u[i-1];  
50     }
```

Figure 9: First loop of **tridag**.

In Figure 9 we can see how each iteration accesses the result of  $u$  from the previous iteration. Hence this is not immediately parallelizable.

```
55     for(i=n-2; i>=0; i--) {  
56         u[i] = (u[i] - c[i]*u[i+1]) / uu[i];  
57     }
```

Figure 10: Second loop of **tridag**.

When looking at the other loop, shown in Figure 10, we can see that each iteration access  $u$  of the following iteration before it is overwritten by its respected iteration. To be able to execute **tridag** in parallel these issues must be dealt with first.

### Inlining tridag

Looking at a profiling of the large dataset we determine what is the bottleneck now the coalescing is not the main factor:

tridag1	650ms
tridag2	764ms
initUandV2Dim	391ms

We see that the tridag functions is the main bottleneck. These perform calculation of `a`, `b`, `c` and `_y`, as well as the sequential version of tridag in each dimension. We observe that each kernel's call to tridag only accesses the row (now transposed) which it just wrote to, eliminating any dependencies. Thus, the arrays `a`, `b` and `c` can be removed in favour of simply calculating the values at the point of use. Values from array `a` and `b` are only read once, so removing these arrays does not come at any cost. Values from array `c` however, are used twice. By benchmarking we determine that it is not worth it to calculate these twice to eliminate the array, so this array is kept.

For tridag2, the same logic is applied to the array `_y`, and so this is also removed. With these changes the runtime of tridag1 and 2 are reduced to around 484ms, for a total runtime on large of 1.35 seconds.

## Conclusion

Having done the most significant transformations, we perform some smaller ones. Some reads and writes scattered throughout the kernel are the same, and the redundancy can be removed by moving the calculation to a single local variable. Performing these changes, and creating a 3D-version of the initUandV kernel to improve parallelism on the small and medium dataset, we obtain our final runtimes:

Dataset	OpenMP time	CUDA time
Small	0.259s	0.088s
Medium	0.323	0.136s
Large	12.1s	1.27s