PMPH project

Malte Stær Nisssen - tgq958
René Løwe Jacobsen - vlx198
Erik John Partridge - xnk343

October 30, 2015

1 Project Specifications

The group project for the course Programming Massively Parallel Hardware is about parallelizing a serial implementation of `tridag`, contained in the files ProjCoreOrig.cpp, ProjectMain.cpp, ProjHelperFun.cpp, and ProjHelperFun.h. The project consists of three parts: converting to OpenMP parallel, converting to CUDA parallel, and results. We will be comparing the CUDA and OpenMP implementation with the original code. These optimizations and translations were done with the help of the slides from the course, listed in the bibliography. [1] [2] [3]

2 OpenMP

2.1 The Idea

Converting serial loops into parallel loops is as simple as ensuring that there are no loop dependencies. We must check that for every loop we wish to parallelize, that no iteration depends on the results of any other iteration, and that variables which belong to multiple iterations have either the correct scope, so that information does not pass from one iteration to the next, or a separate index for each iteration. After that, we denote the loop ready by adding the appropriate `#pragma ...` command, and compiling with the correct options.

2.2 The Work

Converting the original code to OpenMP was a straightforward matter, involving only modifying code in ProjCoreOrig.cpp. After ensuring that none of the inner loops would create race conditions, we moved the `REAL strike` and `PrivGlobs globs` declarations inside the main loop, and then simply added the line

```
#pragma omp parallel for default(shared) schedule(static) if(outer>8)
```

right before the same loop declaration. We decided to add the static scheduling line for more

than 8 threads, since that was what we were working with. After that, all that was left was to compile with `g++` using the command line option `-fopenmp`. The results are displayed in the table on page 9.

2.3 Correctness

The changes we made were very minor, involving moving 2 variable declarations inwards in scope, and adding the `#pragma ...` line. The variable declarations made no difference, since every iteration of the loop resulted in both `strike` and `globs` being immediately assigned new values, but moving them allowed for us to parallelize without worrying if the results of one iteration spilled over into another. The `#pragma ...` declaration signaled that this loop was able to be run in parallel. We can assure that it was by looking through the code. From the outermost loop, there were no loop dependencies from one iteration to the next: the results were saved in separate `res[]` indexes, and no result from a previous iteration was used in a later iteration.

3 CUDA

3.1 Preparations

The code as it stands can be naïvely converted to CUDA, by replacing the inner loops with the exact same function inside a CUDA kernel. We are going to first implement this, and then continue by optimizing the TRIDAG kernel.

3.2 Loop Distribution and Array Expansion

3.2.1 The Idea

The first step was to conglomerate every secondary function, which allowed us to hoist the initialization of the globs (global variables) into glob arrays. That is, instead of initializing a temp variable or array for every iteration of a loop, we instead initialize an array one dimension larger, with size equal to the number of iterations of the loop, before the looping code.

The purpose of this is twofold. First, this allows us to serialize a computation which would otherwise be repeated by every thread. By computing it beforehand, the threads can be spared this extra work. Second, this allows us to easily parallelize the inner loops. Since each iteration of the inner `j` loop requires access to a single `tmpA` per iteration of the `i` loop, we would have to compute it, then pass it on as a variable to each of the threads. In the hoisted version, `tmpA[]` is copied to device memory, so that the inner loop can just access the appropriate version without much trouble.

```
for (int i = 0; i<max; i++){
  float tmpA = 0.0;
  for (int j = 0; j<max2; j++){
    tmpA += 2*B[j];
    ...
  }
  ...
}
```

Figure 1: A code snippet with tmpA initialized for every iteration.

```
float tmpA[max] = {0.0, ...};
for (int i = 0; i<max; i++){
  for (int j = 0; j<max2; j++){
    tmpA[i] += 2*B[j];
    ...
  }
  ...
}
```

Figure 2: The same code with tmpA hoisted.

3.2.2 The Work

The first step for us was to begin preparing the CPU for kernel parallelization. Before distributing the outer loops in ProjCoreOrig.cpp, we began by moving all of the secondary functions (`void updateParams`, `void setPayoff`, `REAL value`, and `void rollback`) into `void run_OrigCPU`. This allowed us to easily see the globs and their relations to one another.

The work at this step is temporary. The code is objectively de-optimized, since more time is spent on initializing arrays, memory usage is larger, and the code runs slower. The purpose of this is to prepare the code for optimal loop distribution.

The `REAL strike` variable has been removed, being placed instead inside of one of the kernels. The variables which have been hoisted at this step are listed in figure 3.

The other necessary modifications were to simply update the relevant variable references, for example, changing `globs.myResult[j][k]` to `globArr[i].myResult[j][k]`.

3.2.3 Correctness

The reason we are allowed to do this is because we are not fundamentally changing anything about the flow of the program. Moving all of the functions together does nothing to program flow, only impeding readability slightly. For the hoisted variables, the extra dimension can be

```
void run_OrigCPU(...
{
  ...
  // Generate vector of globs. Initialize grid and operators onces
  // and make default element of vector
  // Hoisted from "value"
  PrivGlobs globs(numX, numY, numT);
  initGrid(s0, alpha, nu, t, numX, numY, numT, globs);
  initOperator(globs.myX,globs.myDxx);
  initOperator(globs.myY,globs.myDyy);
  vector<PrivGlobs> globArr (outer, globs);
  ...
  //Rollback globs
  vector<vector<vector<REAL> > > u(outer,vector<vector<REAL> >(numY,
     vector<REAL>(numX))); // [outer][numY][numX]
  vector<vector<vector<REAL> > > v(outer,vector<vector<REAL> >(numX,
     vector<REAL>(numY))); // [outer][numX][numY]
  vector<vector<REAL> > a(outer,vector<REAL>(numZ)),
     b(outer,vector<REAL>(numZ)), c(outer,vector<REAL>(numZ)),
     y(outer,vector<REAL>(numZ)); // [outer][max(numX,numY)]
  vector<vector<REAL> > yy(outer,vector<REAL>(numZ));
  // temporary used in tridag
  // [outer][max(numX,numY)]
  ...
```

Figure 3: Hoisted variables in ProjCoreOrig.cpp.

easily compared to a new variable per iteration, and since the variables are moving outwards in scope, nothing vital is changed. There is no danger of loop dependencies, since each iteration still uses their own indexes for these hoisted variables.

After this work, the program is in a state to distribute the various loops. There is a mild amount of slowdown, since we allocate more memory to some of the variables, and we perform some initial calculations which otherwise performed per loop iteration. This is not a good stopping point, but it is necessary to continue.

3.3 Kernel Replacement

3.3.1 The Idea

After loop distribution, we began to convert the loops to CUDA kernels. These kernels are copied almost directly from the already existing CPU code. This allowed us to begin using the GPU, and CUDA.

3.3.2 The Work

For each of the distributed loops, we naïvly translated the code into a CUDA kernel. That is, we attempted to rewrite the functions so that they performed precisely the same task on the GPU as they did on the CPU. Figures 4 and 5, show the conversion of the function `void setPayoff` from ProjCoreOrig.cpp into the `__global__ void setPayoffKernel` in the file ProjKernels.cu.h. All distributed loops and matching functions were converted.

```
void setPayoff(const REAL strike, PrivGlobs& globs )
{
  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;
  }
}
```

Figure 4: The distributed loop implementation of `void setPayoff`.

```
template<const unsigned T>
__global__ void setPayoffKernel(
       const unsigned outer,
       const unsigned numX,
       const unsigned numY,
       REAL* myX,
       REAL* myResult
       )
{
  int i = blockIdx.x*T + threadIdx.x; // outer
  int j = blockIdx.y*T + threadIdx.y; // myX.size
  int k = blockIdx.z*T + threadIdx.z; // myY.size
  if (i < outer && j < numX && k < numY) {
    myResult[i * numX*numY + j * numY + k] = max(myX[i * numX +
        j]-0.001*i, (REAL)0.0);
  }
}
```

Figure 5: The equivalent CUDA kernel.

The next step was to load the appropriate variables to and from memory. figure 6 shows some of the variables that are now on device memory, to be accessed by the kernels. This is analogous to the `globs` structures from the sequential code.

```
// Arrays for rollback:
REAL* d_a, *d_b, *d_c, *d_y, *d_yy; // [outer][max(numX,numY)]
REAL *d_v, *d_u;
cudaMalloc((void**) &d_a, sizeof(REAL)*outer*numX*numY);
cudaMalloc((void**) &d_b, sizeof(REAL)*outer*numX*numY);
cudaMalloc((void**) &d_c, sizeof(REAL)*outer*numX*numY);
cudaMalloc((void**) &d_y, sizeof(REAL)*outer*numX*numY);
cudaMalloc((void**) &d_yy, sizeof(REAL)*outer*numX*numY);
cudaMalloc((void**) &d_u, sizeof(REAL)*outer*numX*numY);
cudaMalloc((void**) &d_v, sizeof(REAL)*outer*numY*numX);
REAL * timeline = (REAL*) malloc(sizeof(REAL)*numT);
cudaMemcpy(timeline,d_globs.myTimeline,sizeof(REAL)*numT,cudaMemcpyDeviceToHost);
```

Figure 6: Initializing device memory.

3.3.3 Correctness

It is no longer as easy to argue that this is a simple replacement. Transforming the distributed loops into kernels requires that the syntax has to change, and the semantics as well. The example in figure 5 shows that the loop iteration has been replaced by `threadIdx`; at a fundamental level, we are moving from sequential loops to parallel threads. The equations are the same, with a slightly more complicated index (`threadIdx` instead of a simple `i`), and the variables are now all loaded into GPU memory.

There are two arguments for correctness. The first is that although the semantics and syntax have both changed, from distributed loops to threads, the process is a one-to-one mapping. Every kernel can be directly traced back to a loop, and every loop gave rise to a single kernel. As long as the input and output of the kernels and loops is the same, we have the same program.

The second is that we continuously validated the GPU kernels during the programming by copying intermediate CPU results to the GPU, calculating the GPU kernel results, and then comparing them to the CPU results. That is, each step was substituted piecewise to ensure that they functioned the same.

Although the program has now been distributed to the GPU, the runtime of the entire program took a hit. The problem was, that the first CUDA call of the program also loads the entire CUDA runtime library, which gives a major overhead compared to the actual running time on our datasets. Timing early versions of our code was highly dominated by the CUDA library load time. To combat this, we have added a `cudaFree(0);` line in the ProjectMain.cu before we start timing.

3.4 TRIDAG

The `tridag` algorithm is a tridiagonal solver, used in systems of continuous differential equations when solved using something known as the Crank-Nicholson finite-difference method.

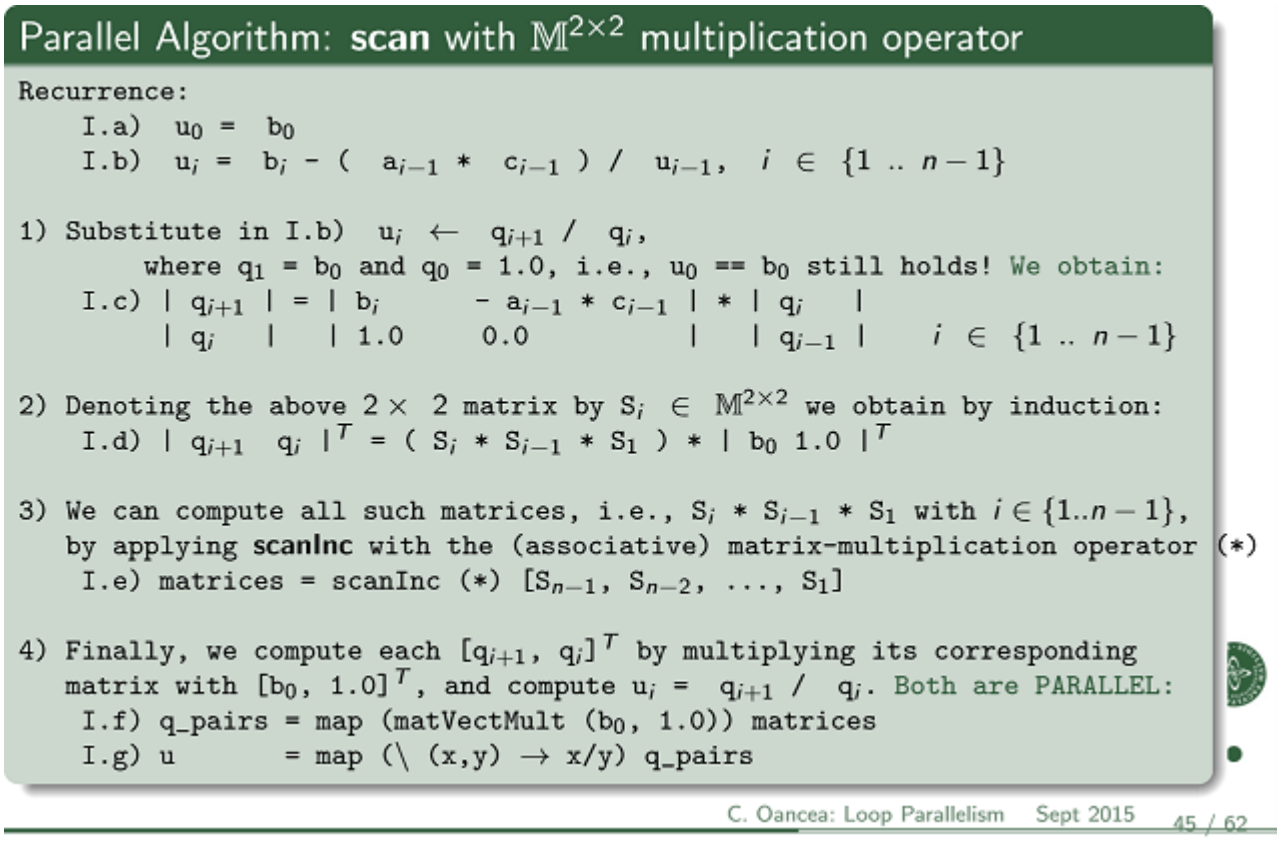
The solution involves large matrix multiplications for the many iterations the finite-differences require, and thus seems like a prime candidate for GPU parallelization. The only problem is that each iteration requires knowledge about the previous iteration, and thus, it is *not* natively fit for parallelization. However, using some variable substitution, we are able to get `tridag` into a form where we can turn the entire algorithm into a series of `scanInc` operations.

3.4.1 The Idea

The original `tridag` implementation worked by finding a solution to the matrix equation

$$A_{n\times n} \times X_{n\times 1} = D_{n\times 1}$$

where the A matrix has the information in a diagonal fashion, D is known, and we are tasked with finding X. Instead, we are going to perform a matrix decomposition, such that $A = L \times U$ where L is a lower diagonal matrix, and U is an upper diagonal matrix. Then, the algorithm is broken into 3 steps. 1: The LU decomposition is calculated. 2: $L \times Y = D$ is solved. 3: $U \times X = Y$ is solved. The algorithm using `scanInc`, which calculates U in step 1, is then as follows, lifted from the slides [1]:



3.4.2 The Work

In addition to the `scanInc` implementation, we also made several optimizations. Many of the changes are minor, and include re-using shared memory. Both `u` and `uu` have been moved to shared memory.

Whenever there were consecutive += operations in `explicitX` and `explicitY`, we created a local variable to hold the value until the final += operation.

Some of the kernels are computed transposed, and some use transposed inputs as compared to the original: `explicitX`, `explicitY`, `implicitX`, and `implicitY`; some of the variables are also transposed: `U`, `A`, `B`, and `C`. The order of these operations is listed in figure 7.

```
1. explicitX -> trU
2. explicitY(trU) -> trU
3. trU->U
4. implicitX->trA,trB,trC
5. trA -> A, trB -> B, trC -> C
6. tridag(U) -> U
7. U -> trU
8. implicitY(trU)
9. tridag
```

Figure 7: The transposed variables and functions, where `trX` means "the transpose of X."

3.4.3 Correctness

Several variables are moved to shared memory, some of the kernels are computed transposed, and some use transposed inputs as compared to the original: `explicitX` calculates `u` transposed; `explicitY` accumulates values into `u` transposed; `implicitX` calculates transposed `a`, `b`, and `c`, followed by transposing the results (so that we receive standard `a`, `b`, and `c`); `implicitY` uses the transposed `u` (`u` is modified by the first `tridag` call before `implicitY` is calculated), so `u` is transposed just before `implicitY` is called.

We are assured of the correctness of these modifications. We know that the variables we moved to shared memory were not going to cause race conditions, since they are accessed as read variables. The transposed/regular variables do not matter as long as the calculations work the same, and for some of the operations it is easier to perform them with a transposed version of the matrices.

Finally, the GPU program was constantly validated against the CPU implementation.

Figure 8: Results of the three different implementations.

4 Results

4.1 Comparisons

4.2 Conclusion

References

[1] Oancea, Cosmin E., and Troels Henriksen. "Loop Parallelism I." PMPH Lecture Notes. DIKU, University of Copenhagen. Sept. 2015. Lecture.

[2] Oancea, Cosmin E., and Troels Henriksen. "Parallel Basic Blocks and Flattening Nested Parallelism." PMPH Lecture Notes. DIKU, University of Copenhagen. Sept. 2015. Lecture.

[3] Oancea, Cosmin E. "Project Related Discussion." PMPH Lecture Notes. DIKU, University of Copenhagen. Oct. 2015. Lecture.