PMPH project

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

October 31, 2015

1 Project Specifications

The group project for the course Programming Massively Parallel Hardware is about parallelizing a serial program, 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 optimizing), and results. We will be comparing the naïve and optimized CUDA and OpenMP implementations 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 (except `tridag`), by replacing the inner loops with the exact same function inside CUDA kernels. We are going to first implement this, optimize the naïve kernels, 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 (`updateParams`, `setPayoff`, `value`, and `rollback`) into `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 `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.

We interchanged the `outer` loop inwards and distributed it, so that all secondary functions got an extra dimension. Giving us the possibility to create 3D kernels later on.

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

```
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.

flow, only impeding readability slightly. For the hoisted variables, the extra dimension can be 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 indices for these hoisted variables.

Distributing the `outer` loop inwards is also allowed as the loop is parallel and a parallel loop can always be interchanged inwards. Distributing the loop is also allowed as all loops inside the `outer` loop will still be inside an `outer` loop.

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 `setPayoff` from ProjCoreOrig.cpp into the `setPayoffKernel` in the file ProjKernels.cu.h. All distributed loops and matching functions were converted.

```
for(unsigned i = 0; i < outer; ++ i) {
      for(unsigned j=0;j<globArr[i].myX.size();++j)
      {
         for(unsigned k=0;k<globArr[i].myY.size();++k)
         {
            globArr[i].myResult[j][k] = max(globArr[i].myX[j]-0.001*i,
               (REAL)0.0); // privatized one level in
         }
      }
   }
```

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

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 `PrivGlobs` structures from the sequential code.

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.

The semantics and syntax have both changed from distributed loops to threads but 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.

```
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.

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 and optimizations

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 maps and scan inclusive operations.

3.4.1 The Idea

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

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

```
// 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.

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. We see the recurrence:

$$u_0 = b_0$$
$$u_i = b_i - \frac{(a_{i-1} * c_{i-1})}{u_{i-1}}, i \in \{1 \ldots n-1\}$$

If we then substitute u_i with $\frac{q_{i+1}}{q_i}$ where $q_1 = b_0$ and $q_0 = 1.0$, then $u_0 = b_0$ still hold and we have:

$$\begin{bmatrix} q_{i+i} \\ q_i \end{bmatrix} = \begin{bmatrix} b_i & -a_{i-1} * c_{i-1} \\ 1.0 & 0.0 \end{bmatrix} * \begin{bmatrix} q_i \\ q_{i-1} \end{bmatrix}, i \in \{1 \ldots n-1\}$$

If then denote the above 2-by-2 matrix by S_i we can then write S_{i+1} as $(S_i * S_{i-1} * S_1$ and then we have this:

$$\begin{bmatrix} q_{i+i} \\ q_i \end{bmatrix} = (S_i * S_{i-1} * S_1) * \begin{bmatrix} b_0 \\ 1.0 \end{bmatrix}$$

All of the matrices $S_i * S_{i-1} * S_1$ with $i \in \{1 \ldots n-1\}$ can then be calculated with a scan inclusive and a matrix multiplication operator. When we have q_{i+1} and q_i, we can then calculate u_i as $u_i = \frac{q_{i+1}}{q_i}$.

3.4.2 The Work

Because of time constraints, we have used the `TRIDAG_SOLVER` given to us instead of implementing our own tridag solver from scratch. In addition to the maps and `scanInc`

optimizations of tridag, we made several optimizations to the naïve CUDA implementation. Many of the changes are minor, and include: re-using shared memory, transpose arrays for memory coalesced access, keep variables in private cache until modification is done before writing to global memory, reduce the number of floating point operations by simplifying computations.

In tridag, we are able to re-use the shared memory for `mat_sh` and `lin_sh` since all computations using `mat_sh` are completed before doing computations on `lin_sh`, and both arrays have one element per thread. Both `u` and `uu` have been moved to shared memory for fewer accessed to global memory. Finally, the `a[gid]` is accessed twice and thus by holding it in a register we save yet another access to global memory.

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

The following changes are performed in order to achieve memory coalesced access in our CUDA kernels:

explicitX is changed to compute the transpose of `u`.

explicitY writes directly to the transpose of `u`.

implicitX computes the transpose of `a`, `b`, and `c`.

tridag is called on `u`, `a`, `b`, and `c`, and thus these four arrays are transposed before calling the first `tridag`.

implicitY needs to use the transpose of `u`. `Tridag` however wrote to `u`, and thus we must transpose the new value of `u`.

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.

4 Results

Table 1 shows the total execution times for the four versions of the code: Sequential CPU, OpenMP CPU, Naïve CUDA, and Optimized CUDA for both the small and large dataset.

We have deliberately excluded the medium dataset since the `tridag` CUDA kernel only runs on datasets matching the dimensions of the CUDA kernel.

Timing was done with the use of the test code provided to use by Cosmin. Since we are interested in the runtime of the algorithm, not the entire program, the only line we changed was to add the `cudaFree(0);` line before starting the timer. This prevents the CUDA library load times from adding to the algorithm.

All versions of the code validate against the Sequential CPU.

Version	Total execution time (microseconds)	
	Small	Large
Sequential CPU	2297659	216305907
OpenMP CPU	213948	10132446
Naïve CUDA	92787	6975193
Optimized CUDA	60867	3647836

Table 1: Results of the three different implementations.

5 Conclusion

The benefits of parallelism are readily apparent in the table. For the small data set, the conversion from Sequential CPU to OpenMP results in the algorithm running over 10 times faster. The conversion from OpenMP to Naïve CUDA gains us another factor of approximately 2, and the Optimized CUDA gets us around another $\frac{1}{3}$. The Optimized CUDA has a speedup of roughly 37 times that of the Sequential CPU.

For the large data set, we can see even bigger speedup. The OpenMP CPU runs 21 times faster than sequential. The Optimized CUDA runs almost 60 times faster than Sequential CPU.

That is a massive speedup, although the tradeoff is that the algorithm is significantly more complex. The sequential implementation has a total of 438 lines of code, while the optimized code is sitting in at 1365 lines. There is a lot more work involved in converting to CUDA. By comparison, the OpenMP CPU change required only a single line added, and two lines moved. It has a total length of 439 lines of code.

The conclusion to be drawn from this is that if you are going to run large algorithms many, many times, then it is absolutely worth it to build and optimize GPU code. However, for an intermediate speedup, converting to OpenMP can provide quite respectable improvements, and depending on the code structure, does not necessarily require a large investment of time to implement. As always, there is a trade-off between coding time and run time, but the OpenMP transition can be very effective.

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.