0.1 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, ProjectMain, ProjHelperFun, and ProjHelperFun.h. The project consists of three parts: converting to OpenMP parallel, converting to CUDA parallel, and results and comparisons. We will be comparing the CUD and OpenMP implementation with the original code.

0.2 OpenMP

0.2.1 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.

0.2.2 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 **??**.

0.2.3 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.

0.3 CUDA

0.3.1 Preparations

The code as it stands can be naïvely

0.3.2 Variable Hoisting

The Idea

The first step is 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.

```
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.

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.

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.

```
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.

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]`.

Correctness

The reason we are allowed to do this is because we are not fundamentally changing anything about the flow of the program. 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 indexes for these hoisted variables. After this work, the program is in a state to distribute the various loops over parallel threads. 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.

0.3.3 Distributing Loops

The Idea

notes while writing

mem consumptin bigger, startup time increase threads, lot more threads, more overhead. It is not meant to be fancy, just as precursor to parallelization.

0.3.4 Simple CUDA

writing notes

0.3.5 TRIDAG