Programming Concurrent Systems: Assignment #4

Due on Monday, December 8, 2014

Alyssa - Ilias

December 9, 2014

Introduction

This assignment asked us to perform experiments using OpenACC. In this report, we present results from the first two parts: first, experiments from accelerating our heat dissipation code, and then some results from the provided matmul code.

Heat dissipation — OpenACC

Solution description

Our solution this time is based on the reference code, due to issues with the extensive use of pointer arithmetic in our original code. We used the adapted version of the reference code from Franz Geiger, which fixes compilation issues with the PGI compiler due to the lack of support for C99 multidimensional variable-length arrays.

We rearranged the code to remove unnecessary smearing iterations from the computation loops, and to move it all into a single function for simplicity.

We copy all three matrices (the conduction data, and the source/destination matrices) in at the start of the main loop. The destination data is largely garbage at this point, and so a small performance improvement could be obtained by only copying in the (smeared) edges, but for simplicity we didn't do thisa.

Listing 1: Main loop

```
#pragma acc enter data copyin(c[0:w*h], dst[0:w*h], src[0:w*h])
for (size_t iter = 1; iter <= p->maxiter; ++iter)
{
        /* per-iteration code */
}
```

We parallelized the main dissipation computation, including the smearing. Note that we use the `present` directive, which avoids an unnecessary copy but also makes the GPU aware of the swapped destination/source pointers.

Listing 2: Dissipation computation

```
#pragma acc parallel loop gang independent
 present(c[0:w*h], dst[0:w*h], src[0:w*h])
for (size_t y = 1; y < h - 1; ++y)
{
        #pragma acc loop worker independent
        for (size_t x = 1; x < w - 1; ++x)
        {
                /* computation */
        }

        /* smearing of borders */
}
```

And we also parallelized the reduction step, using OpenACC's directive:

Listing 3: Reduction using acc directive

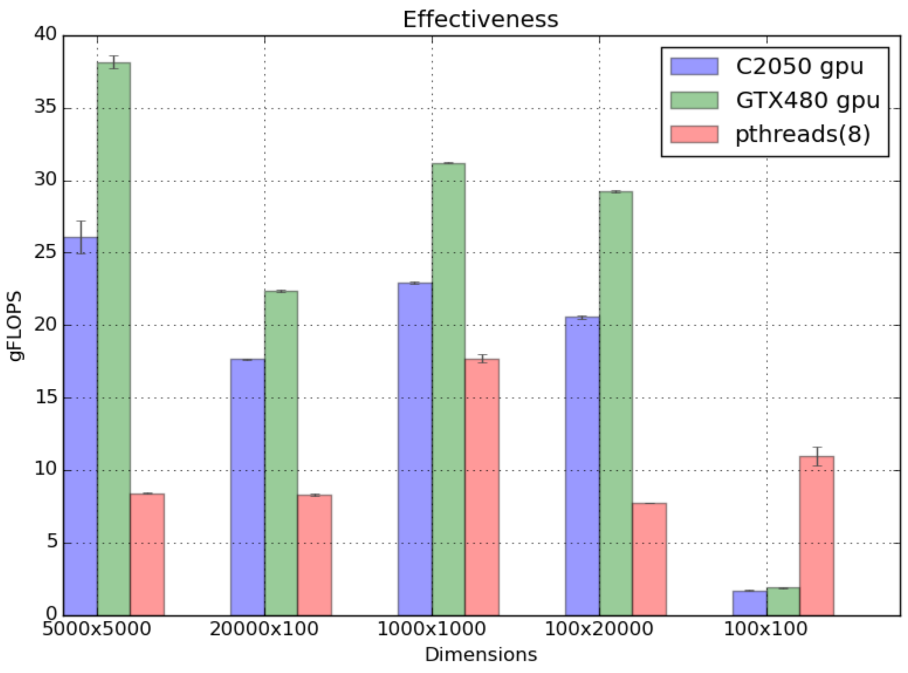
```
// iterate over non-constant rows
#pragma acc parallel loop gang independent reduction(min: tmin)
 reduction(max: tmax) reduction(max: maxdiff)
 reduction(+: tavg) present(dst[0:w*h], src[0:w*h])
for (size_t y = 1; y < h - 1; ++y) {
        #pragma acc loop vectors independent
        for (size_t x = 1; x < w - 1; ++x) {
                /* reduction computation here */
        }
}
```

The use of the `independent` directive on the pragmas informs PGI that the loops do not have data dependencies on each other.

(We also included OpenMP directives, but the resulting code is slower than our original OpenMP code, so we didn't use it.)

Evaluation - Experiments

We run our experiments on the DAS-4 system. For pthreads/OpenMP, we used a normal node which has 8 physical cores, and used 8 threads, which we found to perform best in previous assignments. For the OpenACC results, we used the nvidia (CUDA) backend of the PGI compiler, and ran the result on the DAS-4 systems with either a GTX480 or a Tesla C2050. The limited availability of the nodes with other GPU types made it difficult to experiment with them.

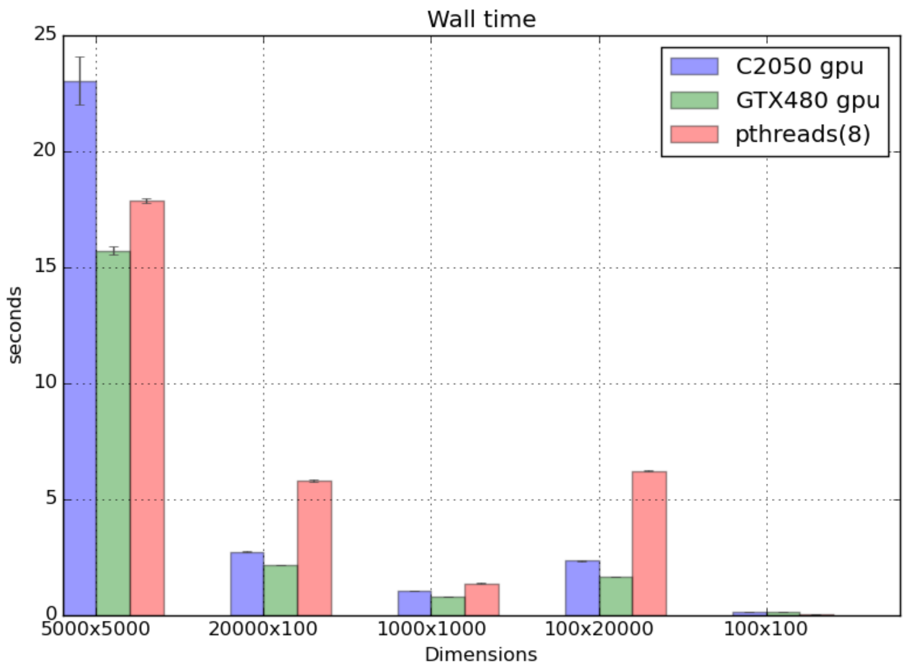


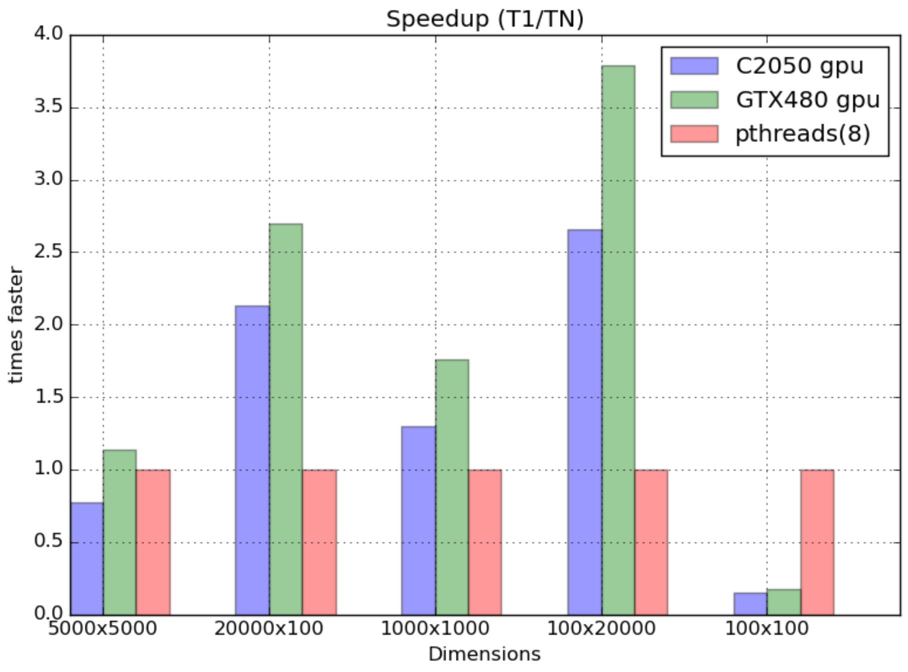
The figure above shows the effectiveness of the parallelisation. The experiments were made with the following parameters:

```
./heat -e 0.0 -i 2000 -k 2001
```

Unsurprisingly, the OpenACC version is generally far superior when using reasonable problem sizes (for the tiny sizes, an `if` clause could be added, similarly to the situation with OpenMP), especially when moving beyond the 'optimal' 1000×1000 size for pthreads/OpenMP and towards large, long-running instances of the problem, since any setup time is eclipsed by the large number of iterations we use here.

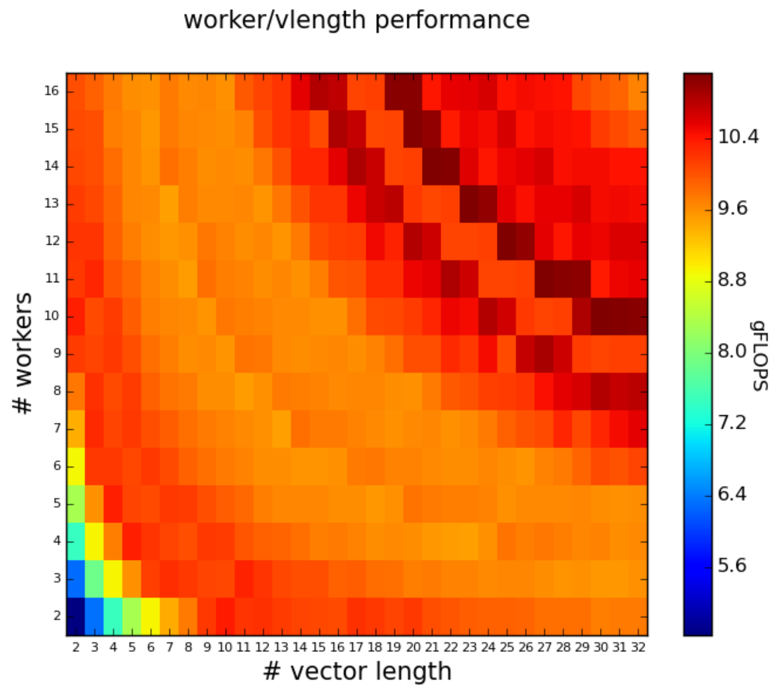
The C2050 has very similar hardware to the GTX480, but with fewer cores, so as expected it is somewhat slower.



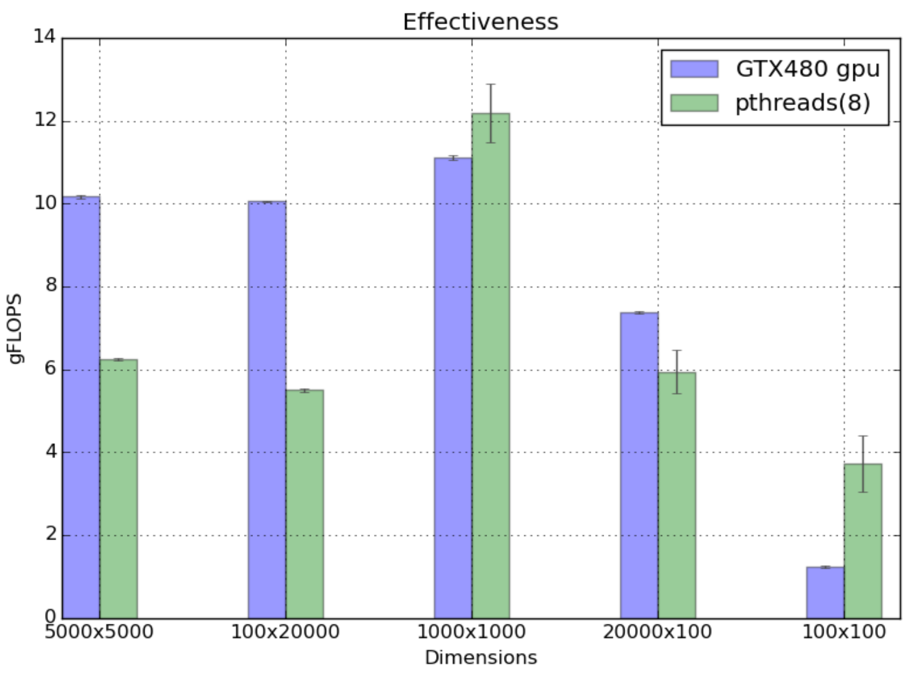


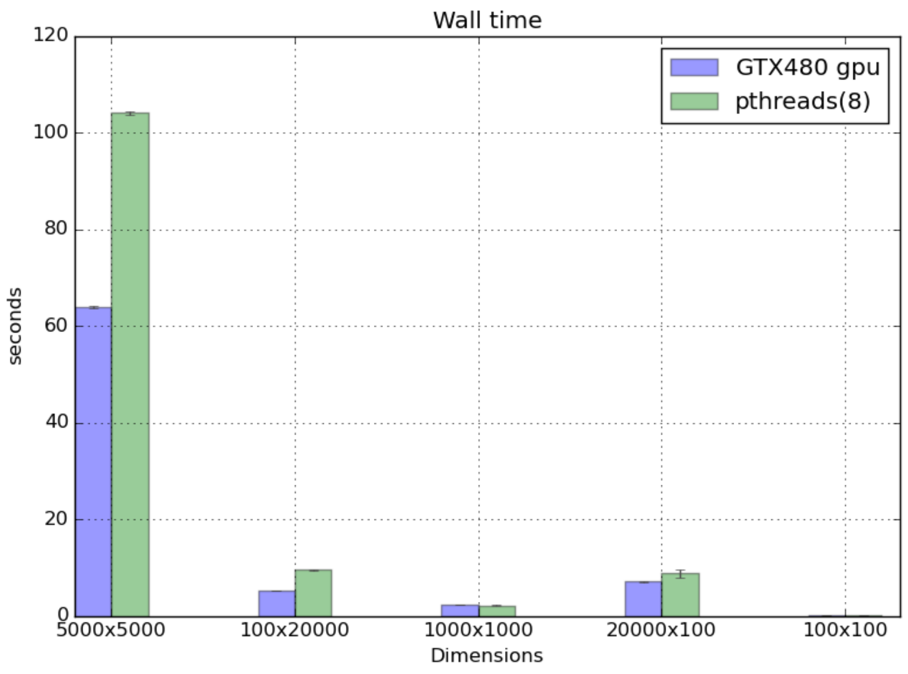
A good GPU implementation of this problem would proceed using tiles/blocks (since the neighbours which are needed by each computation are in all directions), ideally copying each block into shared memory rather than accessing global memory repeatedly. Unfortunately, PGI doesn't support the OpenACC 2.0 `tile` pragma, and in any case, the details of shared memory are not exposed.

We did, however, perform some experiments to try working out the optimal combination of gang, worker and vector sizes to use. Unfortunately, these results were invalidated shortly before submitting this report when we realised that we'd made a fundamental mistake in the loop iterations (we were iterating over the horizontal direction in the outer loop, rather than the vertical one), but we present an example graph showing an example portion of our results from these experiments anyway (PGI assigns the workers/vectors to dimensions of `threadIdx`, so the symmetric nature of the graph is to be expected):

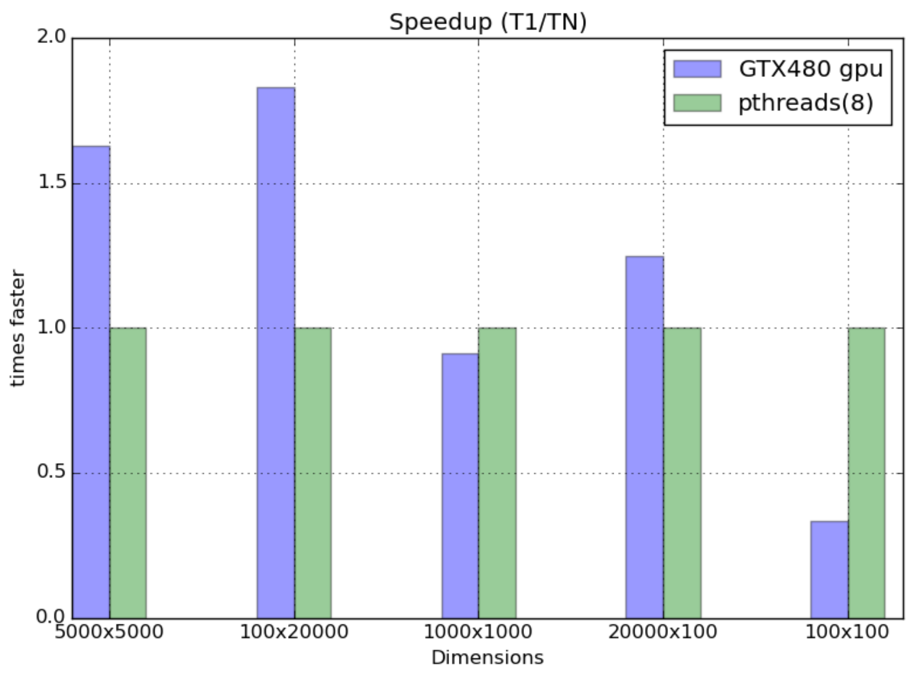


The same mistake is present in the 3 charts below, showing the effectiveness, walltime and speedup of the code with reductions at every step (`-k 1`):



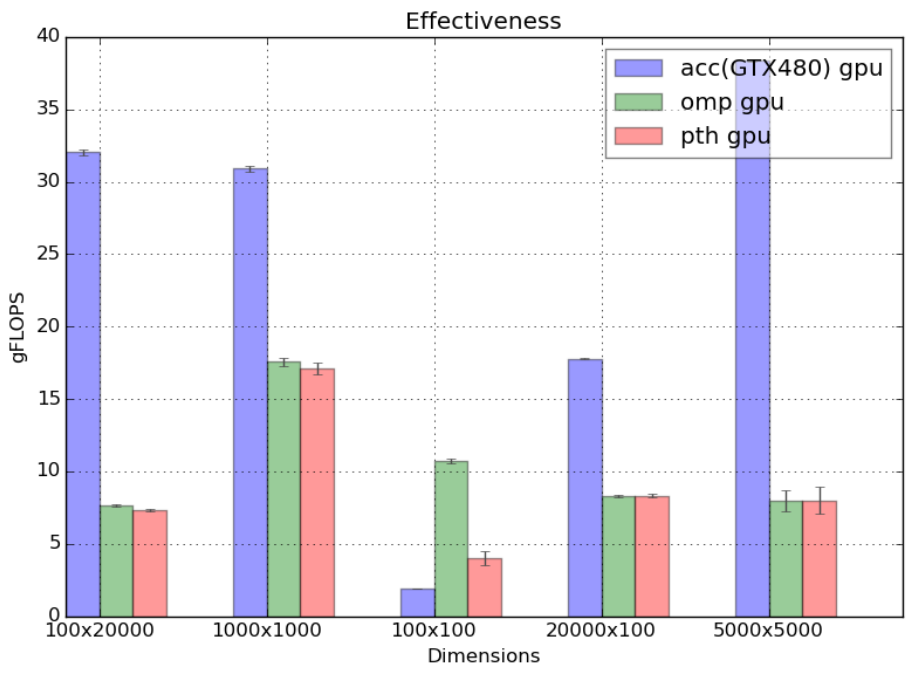


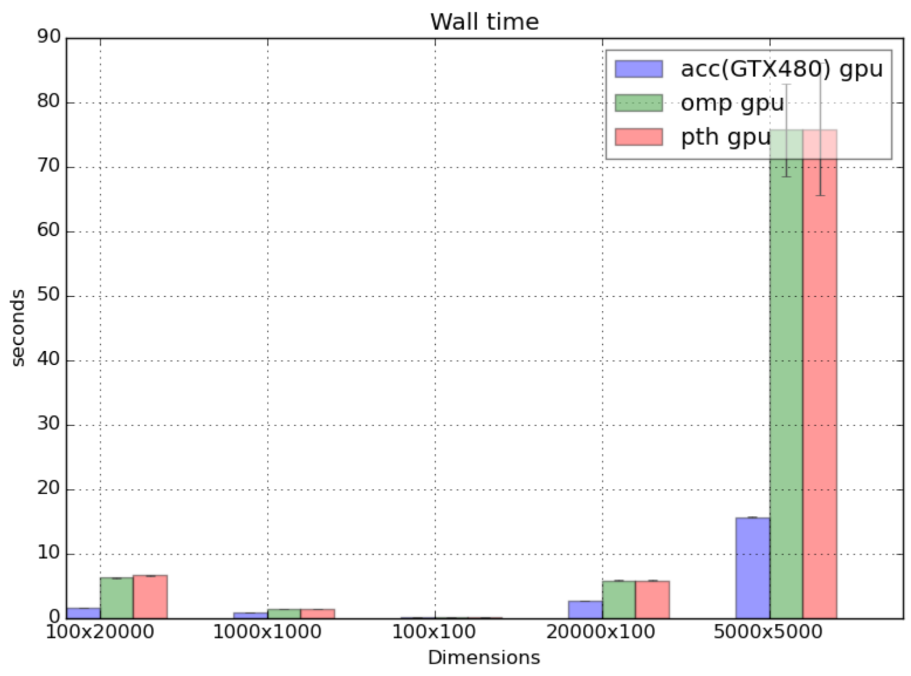
Heat dissipation — OpenACC continued on next page. . .

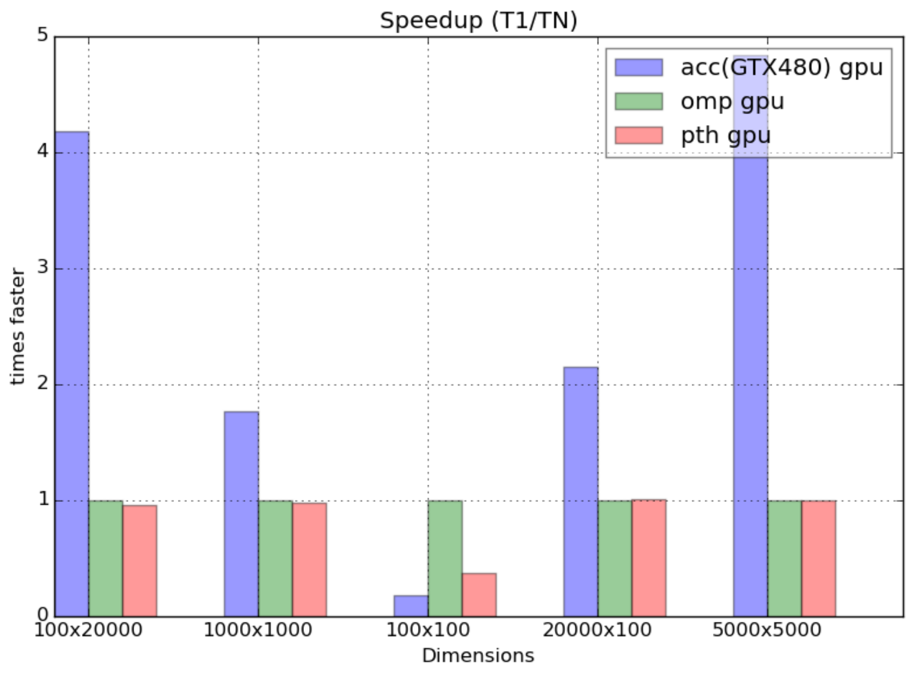


After fixing the mistake we produced the following graphs:

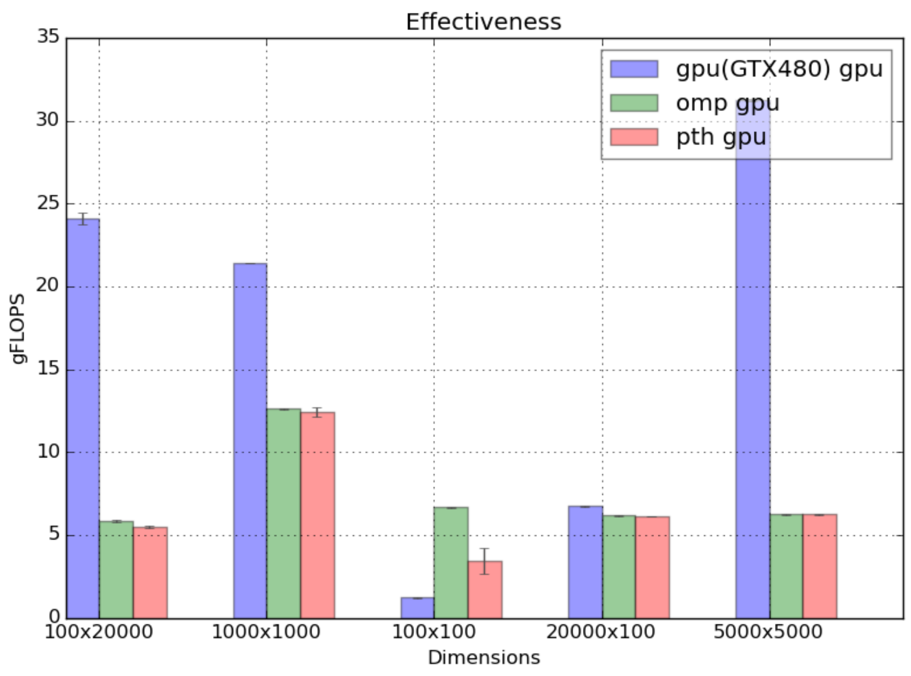
The first three depict a comparison between pthreads, openMP and openACC without doing reductions

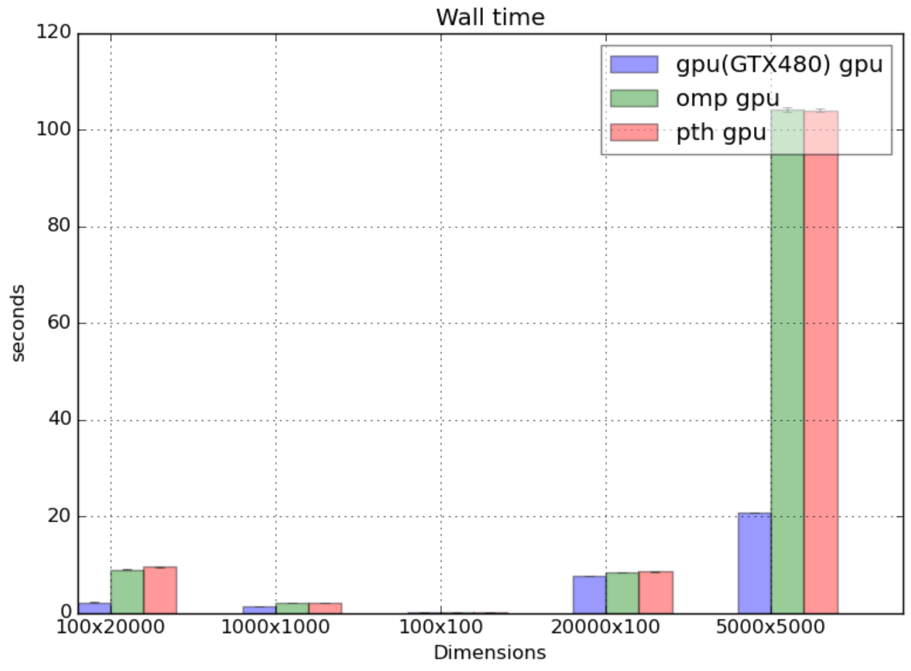


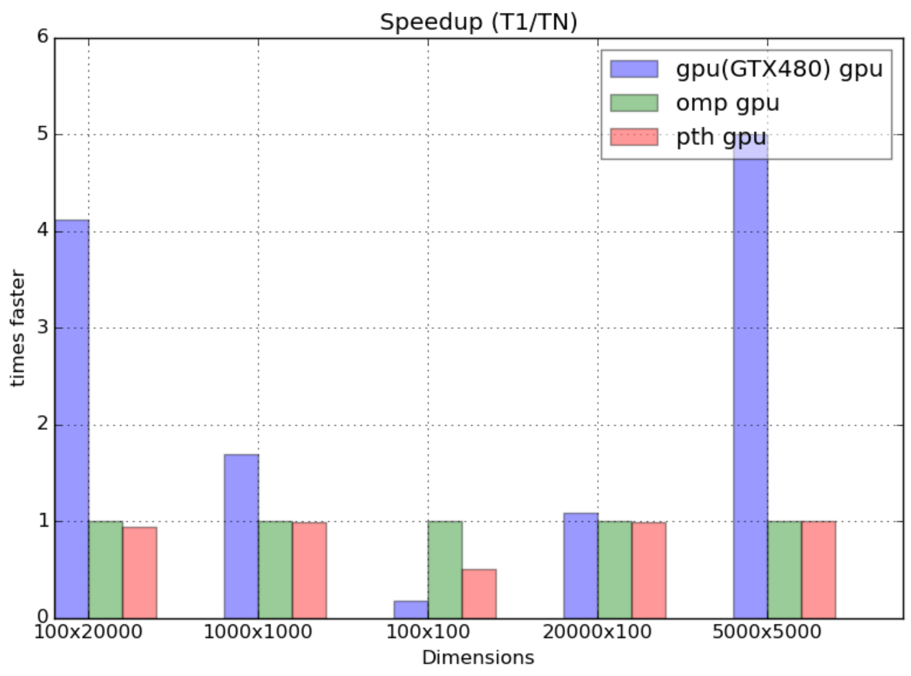




Same comparison with reductions at every step (`-k 1`):


Effectiveness
gpu(GTX480) gpu
omp gpu
pth gpu
gFLOPS
0
5
10
15
20
25
30
35
100x20000
1000x1000
100x100
20000x100
5000x5000
Dimensions


Wall time
gpu(GTX480) gpu
omp gpu
pth gpu
seconds
0
20
40
60
80
100
120
100x20000
1000x1000
100x100
20000x100
5000x5000
Dimensions



Discussion

We can use `pgaccelinfo` to investigate attributes such as the warp size, maximum block size, available shared memory, etc, which we didn't yet find particularly useful during this assignment due to the limitations of OpenACC. Some example output, from a node with a GTX480:

```
Total Constant Memory:         65536
Total Shared Memory per Block: 49152
Registers per Block:           32768
Warp Size:                     32
Maximum Threads per Block:     1024
Maximum Block Dimensions:      1024, 1024, 64
Maximum Grid Dimensions:       65535 x 65535 x 65535
```

To make sure our compute code is compiled optimally, we consider the `-Minfo` output, which gives us information about where Tesla code is generated (and when loops are not vectorized). For example, for the reduction loop (the outer loop is on line 124, and the inner loop on line 127):

```
    124, Generating present(dst[:?])
         Generating present(src[:?])
         Accelerator kernel generated
        125, #pragma acc loop gang /* blockIdx.x */
        127, #pragma acc loop vector(256) /* threadIdx.x */
        132, Min reduction generated for tmin
        135, Max reduction generated for tmax
        138, Sum reduction generated for tavg
        141, Max reduction generated for maxdiff
    124, Generating Tesla code
    127, Loop is parallelizable
```

We used the `PGI_ACC_TIME` environment variable to confirm that our code was running correctly (for example, that it was only reaching the `enter data` region once). The output agrees with this, and justifies our earlier decision not to consider copying in portions of matrices (or using `create` and doing the work on the GPU) due to the massive differences in time between the first data region (copying data in) and the first compute region. It also shows that the computation times for the blocks are largely balanced, with only small differences between them. One example (for a 5000×5000 matrix, and the above parameters) is provided below:

```
71: data region reached 1 time
    71: data copyin transfers: 36
         device time(us): total=421 max=17 min=3 avg=11
86: data region reached 2000 times
86: compute region reached 2000 times
    86: kernel launched 2000 times
        grid: [5000]  block: [128]
         device time(us): total=15,220,640 max=7,638 min=7,600 avg=7,610
```

We can get some insight into how the problem is being split up by PGI by considering the grid/block sizes above, but also more interestingly, for the reduction, which is done very simplistically, using a reduction kernel launched after a main kernel. Again, from the same example:

```
124: kernel launched 1 time
    grid: [5000]  block: [256]
     device time(us): total=3,105 max=3,105 min=3,105 avg=3,105
124: reduction kernel launched 1 time
    grid: [4]  block: [256]
     device time(us): total=25 max=25 min=25 avg=25
```

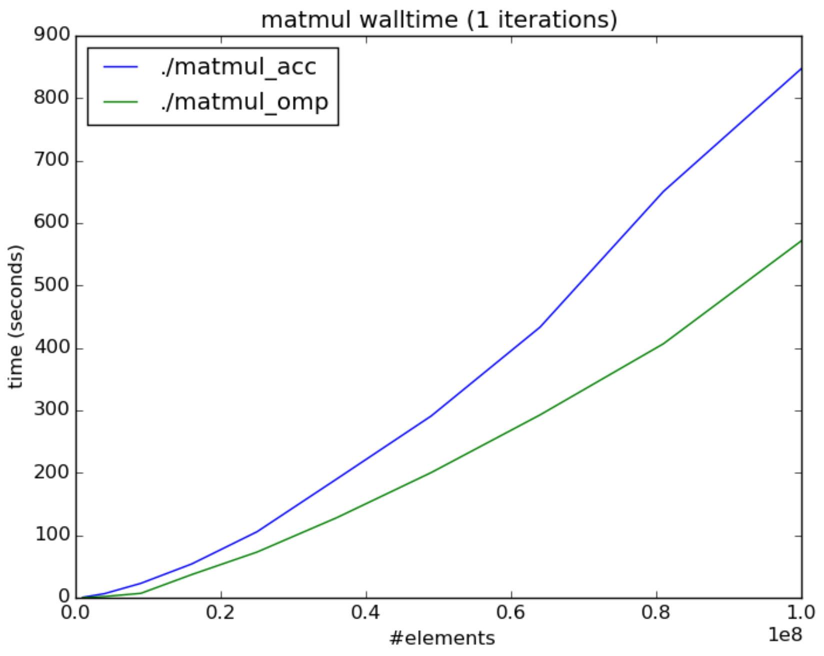
`PGI_ACC_DEBUG` gives somewhat similar (but more detailed) information, although without the timings.

Matmul

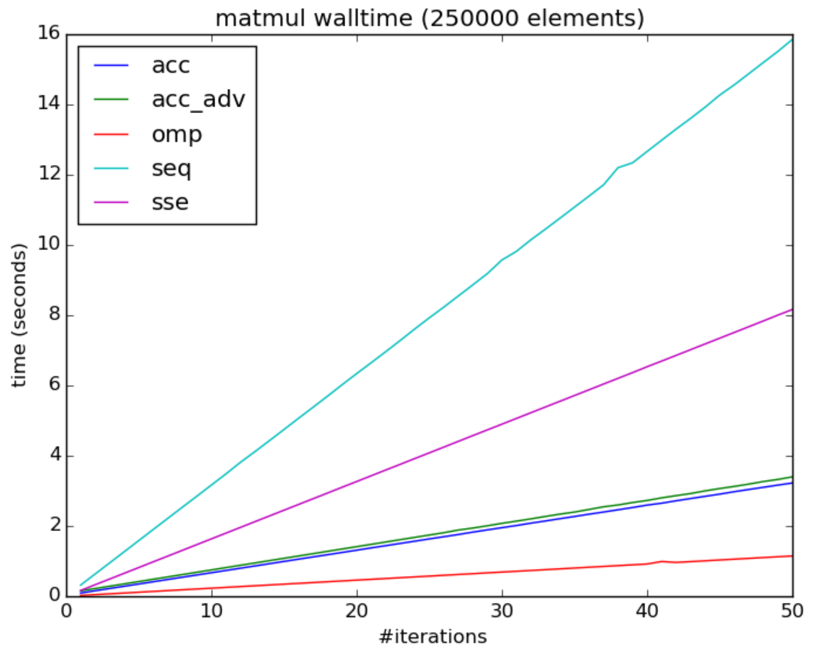
We discussed a lot of the suggestions in the matmul portion of the assignment (e.g., `PGI_ACC_TIME`) above, and since the code was provided for us (so we don't have to worry about optimising sizes or pragmas), we won't discuss these details again for the less interesting matmul case, other than to take a glance at the timing needed for the transfers. For huge sizes or larger number of iterations, the transfers were not significant, but for tiny sizes like $n = 100$ (10,000 elements), the copying eclipses the computation time for a single iteration:

```
45: data copyin transfers: 4
     device time(us): total=36 max=18 min=5 avg=9
67: data copyout transfers: 1
     device time(us): total=14 max=14 min=14 avg=14
48: kernel launched 1 time
    grid: [40]  block: [256]
     device time(us): total=24 max=24 min=24 avg=24
61: kernel launched 1 time
    grid: [40]  block: [256]
     device time(us): total=5 max=5 min=5 avg=5
```

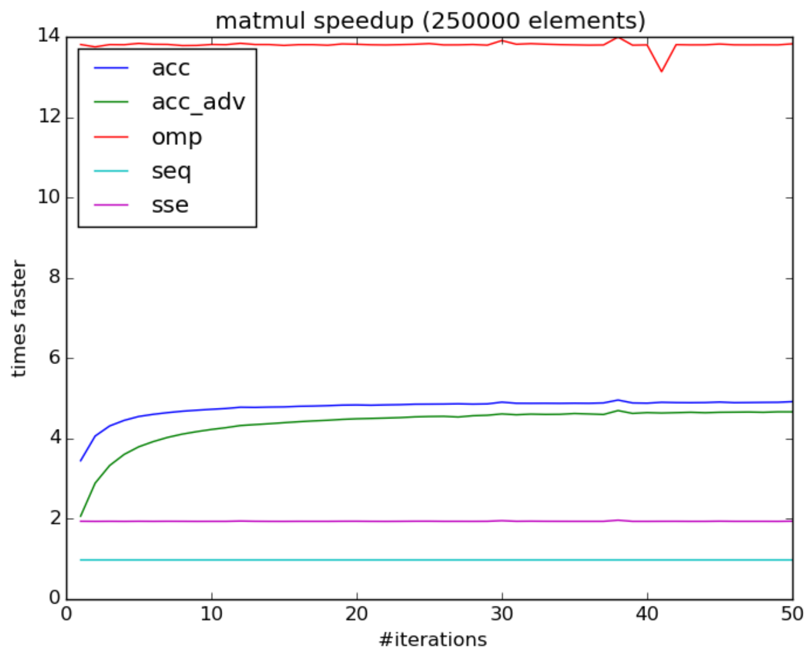
You can observe that, if we only perform one iteration, then the time taken for the multiplication scales reasonably well for the number of elements with both OpenACC and OpenMP, but OpenMP is considerably faster (you can also see the overhead for the OpenACC copy with the tiny sizes):



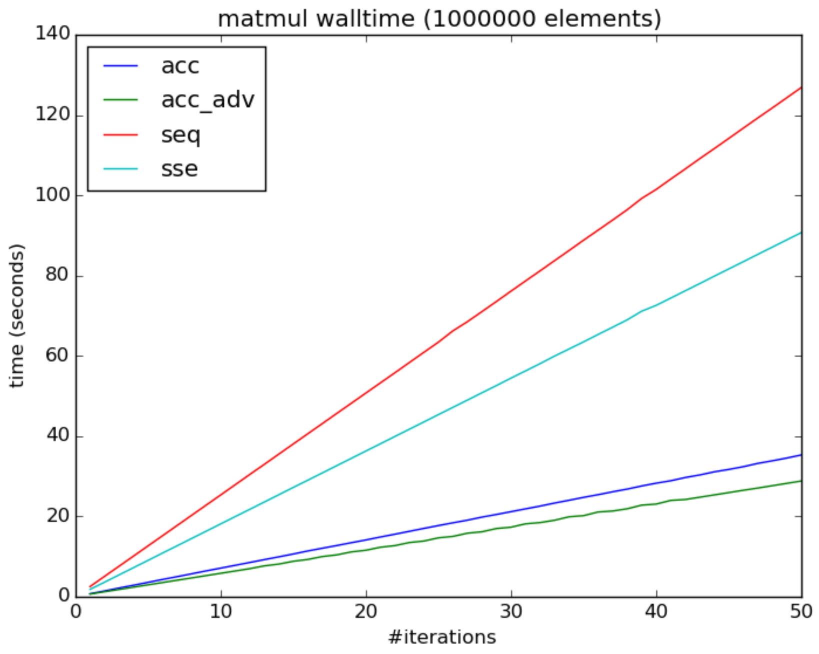
And this is similar for a far larger number of elements:



Matmul continued on next page...



And even with a huge number of elements, the situation is similar. OpenACC is (reassuringly) clearly much better than at least the sequential versions:


matmul walltime (1000000 elements)
acc
acc_adv
seq
sse
time (seconds)
#iterations