HPC ASSIGNMENT 2 – CUDA PARALLELIZATION

GROUP 3 – STEFANO PIGOZZI, CATERINA GAZZOTTI AND FABIO ZANICHELLI

Some changes from last time…

- We noticed the algorithm wasn't allocating memory correctly when computing $\mathbf{A^TAX = Y}$!
  - It did not account for non-square matrices, segfaulting if NX ≠ NY!
  - We fixed all these problems before proceeding to the optimization

- We noticed that Jeston Nano cannot perform atomic adds with double variables
  - So, execution is performed using float variables

```
/* Variable declaration/allocation. */
POLYBENCH_2D_ARRAY_DECL(A, DATA_TYPE, NX, NY, nx, ny);
POLYBENCH_1D_ARRAY_DECL(x, DATA_TYPE, NY, ny);
POLYBENCH_1D_ARRAY_DECL(y, DATA_TYPE, NY, ny);
POLYBENCH_1D_ARRAY_DECL(tmp, DATA_TYPE, NX, nx);
```

kernel_atax optimization with CUDA

- Each thread adds Y vector to A matrix
- First lines of the function find the correct index's number of the current thread
- The atomicAdd at the end of the function doesn't work with doubles
- If HPC_USE_CUDA macro is not defined then sequential code is executed

```
#ifdef HPC_USE_CUDA
__global__ static void kernel_atax_cuda(DATA_TYPE* A, DATA_TYPE* X, DATA_TYPE* Y)
{
    // Find out how many threads there are
    unsigned int threads = gridDim.x * blockDim.x;

    // Find how many iterations should be performed by each thread
    unsigned int perThread = NX / threads + 1;

    // Find the index of the current thread, even if threads span multiple blocks
    unsigned int blockThreadIdx = blockIdx.x * blockDim.x + threadIdx.x;

    // Have each thread perform the previously determined number of iterations
    for(int stride = 0; stride < perThread; stride++)
    {
        // Iterate over x; y is not parallelized
        unsigned int x = threads * stride + blockThreadIdx;

        // Prevent the thread from accessing unallocated memory
        if(x < NX)
        {
            // The same tmp as earlier
            DATA_TYPE tmp = 0;

            for (unsigned int y = 0; y < NX; y++)
            {
                tmp += A[a_index(x, y)] * X[y];
            }

            for (unsigned int y = 0; y < NX; y++)
            {
                // THIS DOES NOT WORK ON THE NANO, AS IT IS TOO OLD TO SUPPORT ATOMIC ADDITION WITH DOUBLES!
                // If you want to use the Nano, swap this for something else, or change atax.hu to use float instead of double
                atomicAdd(&Y[x], A[a_index(x, y)] * tmp);
            }
        }
    }
}
#endif
```

`init_array` optimization with CUDA

```
#ifdef HPC_USE_CUDA
__device__ static void init_array_cuda_x(DATA_TYPE* X, unsigned int threads)
{
    // Find how many iterations should be performed by each thread
    unsigned int perThread = NY / threads + 1;

    // Find the index of the current thread, even if threads span multiple blocks
    int blockThreadIdx = blockIdx.x * blockDim.x + threadIdx.x;

    // Have each thread perform the previously determined number of iterations
    for(int stride = 0; stride < perThread; stride++)
    {
        // Find the index of the current iteration
        // This is equal to `y` of the init_array function
        unsigned int iterationIdx = threads * stride + blockThreadIdx;

        // Prevent the thread from accessing unallocated memory
        if(iterationIdx < NY)
        {
            // Set the array element
            X[iterationIdx] = iterationIdx * M_PI;
        }
    }
}
#endif
```

This function is similar to the initialization of y

```
#ifdef HPC_USE_CUDA
__device__ static void init_array_cuda_a(DATA_TYPE* A, unsigned int threads)
{
    // Find how many elements should be written in total
    unsigned int elements = NX * NY;

    // Find how many iterations should be performed by each thread
    unsigned int perThread = elements / threads + 1;

    // Find the index of the current thread, even if threads span multiple blocks
    int blockThreadIdx = blockIdx.x * blockDim.x + threadIdx.x;

    // Have each thread perform the previously determined number of iterations
    for(int stride = 0; stride < perThread; stride++)
    {
        // Find the index of the current iteration
        // This is equal to `y` of the init_array function
        unsigned int iterationIdx = threads * stride + blockThreadIdx;

        // Determine current x and y
        unsigned int y = iterationIdx % NY;
        unsigned int x = iterationIdx / NY;

        // Prevent the thread from accessing unallocated memory
        if(iterationIdx < elements)
        {
            // Set the array element
            A[iterationIdx] = (DATA_TYPE)(x * (y + 1)) / NX;
        }
    }
}
#endif
```

What's the speedup?

- The optimized version takes 0,278 seconds to execute all the program with large dataset

- So, the speedup is $\frac{old\ time}{new\ time} = \frac{0{,}746}{0{,}278} = 2{,}68$

- Where do we achieve this speedup?

Profiling

- It is better to optimize the `kernel_atax` part or the initialization?
  - The `kernel_atax` part, with large dataset, takes 0,35s with sequential code
  - The optimized version takes 0,26s, so the speedup is 1,35

  - The initialization part takes 0,396s to execute with sequential code
  - The optimized version takes 0,0178s with a speedup of 21,9

- The initialization part generates much more speedup, but also `kernel_atax` generates some speedup

→ **So, the best choice is to optimize both parts**

Experiments on other datasets

	Mini dataset	Small dataset	Standard dataset	Large dataset	Extralarge dataset
Sequential times	$1{,}27 * 10^{-5}$s	0,00344s	0,188s	0,746s	1,68s
Optimized times	$1{,}61 * 10^{-3}$s	0,0112s	0,0647s	0,278s	0,665s
Speedup	**0,0079**	**0,31**	**2,90**	**2,68**	**2,52**

*Speedups written in red are slowdowns

OpenMP **vs** CUDA

	Speedup with OpenMP*	Speedup with CUDA
Mini dataset	**0,476**	0,0079
Small dataset	**3,32**	0,31
Standard dataset	**3,36**	2,90
Large dataset	**3,37**	2,61
Extralarge dataset	**3,47**	2,53

*OpenMP programs are executed with float variables too.
For mini dataset there is no best option than sequential code; OpenMP has only less slowdown

Thanks for the attention!