Parallel Programming
GPUs: OpenACC
2025-26

Roger Pieres Termes. NIU: 1653925
Noel Sánchez Arrieta. NIU: 1636380
Group 11

1 OpenACC: Parallelization of the Laplace code

We will use the following compilation instruction given a `name.c` file.

```
nvc -fast -acc -tp=nehalem -gpu=cc80 -Minfo=accel name.c -o name
```

This instruction will be core for incrementally building our parallelized version of the code. That is because this instruction yields a message that helps to parallelize in the GPU the different loops we encounter. It both recommends next steps for implementing parallelism and helps to correct errors one by one. Hence, it is extremely useful for building robust and reliable code while understanding what is going on.

The first parallelized version we have performed is `lap.c` file. We will compare it to the baseline `laplace2.c` file. Let us discuss the modifications done to the initial provided file for the homework.

2 Modifications to the code

Preliminar functions

Observe that initially we had some declared functions, some of which need to be modified. We kept the `stencil` function that computes the grid calculation for the Laplace algorithm.

laplace_step

However, we've modified the the `laplace_step` function int the following way:

```c
void laplace_step ( float *in, float *out, int n, int m )
{
  #pragma acc data present(in,out)
  #pragma acc kernels
  {
      int i, j;
      #pragma acc loop independent
      for ( i=1; i < m-1; i++ )
        #pragma acc loop independent
        for ( j=1; j < n-1; j++ )
          out[j*m+i]= stencil(in[j*m+i+1], in[j*m+i-1], in[(j-1)*m+i], in[(j+1)*m+i]);
  }
}
```

The instruction `data present` let's the compiler know that the data structures for saving `in` and `out` are already created. Then, we create a parallel region in the GPU using `kernels` with two independent loops. Note that we can do that because the update only depends on the `input` values an not among themselves.

2.0.1 laplace_error

First, we assure no memory allocation is performed twice with `data present` and create a parallel region with `parallel loop`.

```c
float laplace_error ( float *old, float *new, int n, int m )
{
  int i,j;
  float error = 0.0f;

  #pragma acc data present(old,new)
  #pragma acc kernels
  {
    for (i=1; i<m-1; i++)
      for (j=1; j<n-1; j++)
        error = fmaxf(error, sqrtf(fabsf(old[j*m+i] - new[j*m+i])));
  }
 return error;
}
```

If we look into our baseline `laplace2.c`, provided in the virtual campus, we will see that the programmer decides to let the compiler decide how to perform the parallelization with the `kernel` directive. Note, however, that this could create a race condition because all the threads are writing in the same variable `error`. This could be solved by a `reduction(max:error)` instruction to let the GPU know where to write the result. Nonetheless, we have decided to keep it that way, despite being incorrect, because the goal is to beat the baseline performance and the reduction takes substantially more time.

2.0.2 laplace_copy

We do the same as the `laplace_step` with two independent loops. It could also be parallelized with `parallel loop` and `collapse(2)` as the other ones could. We will not do anything to the `laplace_init` function as it is only executed once in the CPU.

2.1 Main Loop

For the main loop we have just created a *data region* to perform our computations. At entry, we copy the information of `A` from host to GPU and create a variable of the same size called `Anew`. These two will live in the GPU inside the data region.

```c
# pragma acc data create(Anew[:n*m]) copy(A[:n*m])
while ( error > tol && iter < iter_max )
{
iter++;

laplace_step (A, Anew, n, m);

error = laplace_error (A, Anew, n, m);
laplace_copy (Anew, A, n, m);
if (iter % (iter_max/10) == 0) printf("%5d, %0.6f\n", iter, error);
}
```

Because we have set the `data present` no expensive data transfers (with the PCI express) happen inside this loop. After it finishes, it copies from GPU to host again matrix `A` and deallocates `Anew`.

3 Performance of different versions

If we execute the provided `lap.c` we can see that it does not improve the baseline time, around 7.5 seconds, for size $(4096, 4096)$, `iter_max=10000` and `tol=1e-3`. For compiling the code we do

```
nvc -fast -acc -tp=nehalem -gpu=cc80 -Minfo=accel lap.c -o lap
```

And we execute with the provided files

```
sbatch -p cuda-ext.q -w aolin-gpu-3 --gres=gpu:1 -o ResGPU3.txt runGPU.txt lap
```

This yields the same output as the baseline taking around the same time. In our trial, it took around 7.3 seconds. In order to beat that, we can look into the profiling information obtained in the `ResGPU3.txt` file. Thanks to that, we can see that the main problem is memory bandwidth. Also, some improvements can be made regarding `sqrt` computation and pointer switching.

lap1.c

The next improvemnt has been done in the `lap1.c` file. One of the first problems we can see in the profiling is the inefficient use of several parallel regions. To solve that, we have united the `error` and `step` functions into one:

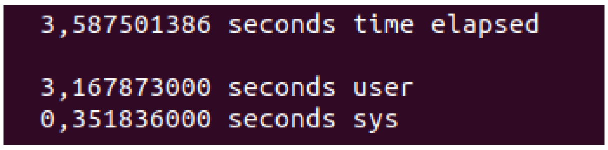
```c
float laplace ( float *in, float *out, int n, int m )
{
    float error = 0.0f;
    #pragma acc parallel loop collapse(2) reduction(max:error) present(in[0:n*m], out[0:n*m])
    for ( int j = 1; j < n-1; j++ )
        for ( int i = 1; i < m-1; i++ ) {
        int idx = j*m + i;

        float newval = stencil(in[idx+1], in[idx-1], in[idx-m],in[idx+m]);
        out[idx] = newval;
        float diff = fabsf(newval - in[idx]);
        error = fmaxf(error, diff);
        }
  return error;
}
```

In addition, we have omitted the `sqrt` from the calculations. Moreover, in the main loop we have eliminated the inefficient copy for a pointer switch.

```c
    float *tmp = cur;
    cur = next;
    next = tmp;
```

In this case, the performance increases substantially. For the same values of tolerance (setting it to `1e-6`) and iterations



```
  3,587501386 seconds time elapsed

  3,167873000 seconds user
  0,351836000 seconds sys
```

Figure 1: Time for the lap1.c version implementation.

Hence, we achieve more than a **2.15** speed-up improvement with this last implementation. This shows the power of a good `C` implementation together with an efficient parallelization.

4 Conclusions

Starting from the baseline GPU Laplace solver (`laplace2.c`), which runs in about 7.2–7.5 seconds for $n = m = 4096$ and `iter_max` = 10000, we developed a dynamic-memory implementation based on `lap.c` and introduced a set of optimizations in the `lap1.c` file to improve performance. Profiling showed that the dominant limitation of the original accelerated version was memory bandwidth: most of the execution time was spent performing redundant full-grid traversals and launching multiple kernels per iteration rather than executing expensive arithmetic. Guided by these results, the most effective improvements came from restructuring the algorithm to reduce memory traffic and kernel overhead. In particular, merging the relaxation update and error computation into a single OpenACC parallel kernel with a `reduction(max:error)` removed an extra sweep over the grid and ensured correct parallel behavior. Additionally, eliminating the `sqrtf` operation in the convergence calculation reduced unnecessary arithmetic work in the innermost loop. Finally, replacing the full-array copy between iterations by pointer swapping removed an entire memory pass per iteration, which is especially beneficial for large meshes. Altogether, these modifications implemented in `lap1.c` produced a final speed-up of more than 2.15× compared to the initial accelerated implementation. Performance depends mainly on minimizing data movement and kernel launches rather than on increasing raw parallelism.