HPC Assignment 2
Durbin optimization with CUDA

https://github.com/Il-castor/hpc-lab-assignment-1

Balma Giovanni; Castorini Francesco; Rossi Lorenzo

From rewritten code to CUDA kernels

```
for (k = 1; k < _PB_N; k++)
{

  beta = beta - alpha * alpha * beta;

  sum = 0;                              kernel 1
  for (i = 0; i <= k - 1; i++)
    sum += r[k - i - 1] * y[(k - 1) % 2][i];
                              reduction (sum)
  sum += r[k];
  alpha = -sum * beta;                  kernel 2
  for (i = 0; i <= k - 1; i++)
    y[k % 2][i] = y[(k - 1) % 2][i] + alpha * y[(k - 1) % 2][k - i - 1];
  y[k % 2][k] = alpha;
}
```

Dividing into CUDA kernels

kernel1 ⟹

```
for (i = 0; i <= k - 1; i++)
  sum += r[k - i - 1] * y[(k - 1) % 2][i];

__global__ static void durbin_k1(int k, DATA_TYPE *r,
        DATA_TYPE *y, DATA_TYPE  *tmp) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < k)
    tmp[i] = r[k - i - 1] * y[i];
}
```

kernel2 ⟹

```
__global__ static void durbin_k2(int k, DATA_TYPE *r,
        DATA_TYPE *yi, DATA_TYPE *yo, DATA_TYPE alpha) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < k)
    yo[i] = yi[i] + alpha * yi[k - i - 1];
  if (i == k)
    yo[k] = alpha;
}

for (i = 0; i <= k - 1; i++)
  y[k % 2][i] = y[(k - 1) % 2][i] + alpha * y[(k - 1) % 2][k - i - 1];
```

Calling kernels

```
for (unsigned int k = 1; k < _PB_N; k++)
{
  beta = beta - alpha * alpha * beta;

  durbin_k1<<<dimGrid, dimBlock>>>(k, r, ycurr, red1);
  gpuErrchk(cudaPeekAtLastError());
  DATA_TYPE sum = reduceCuda(red1, red2, k); ??

  DATA_TYPE rk;
  gpuErrchk(cudaMemcpy(&rk, &r[k], sizeof(DATA_TYPE), cudaMemcpyDeviceToHost));
  sum += rk;
  alpha = -sum * beta;

  durbin_k2<<<dimGrid, dimBlock>>>(k, r, ycurr, ynext, alpha);
  gpuErrchk(cudaPeekAtLastError());
  std::swap(ycurr, ynext);
}
```
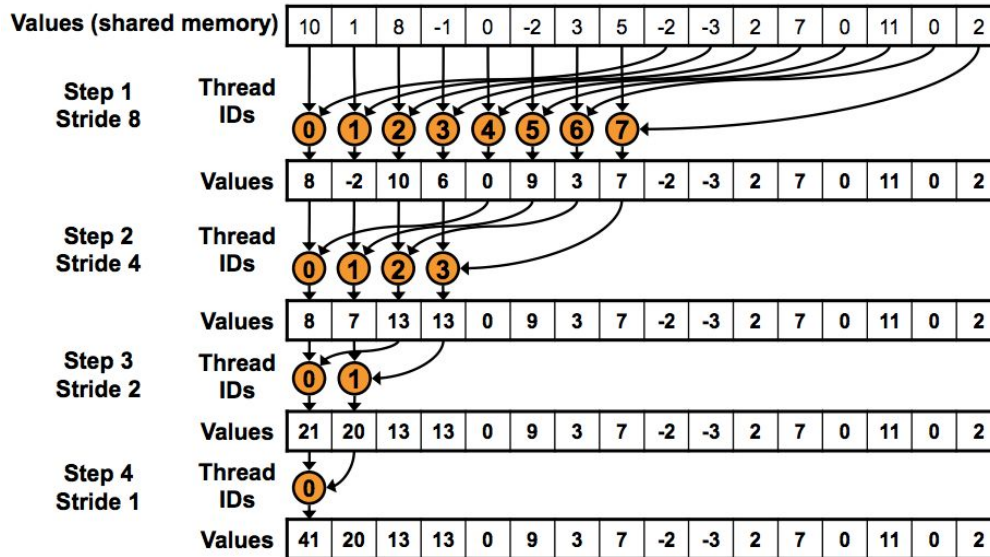
Sum reductions in CUDA

Atomics? Not supported for doubles in Jetson Nano (and no fun!)

Every block sums BLOCK_SIZE elements into one single element using shared memory to synchronize.

Values (shared memory)	10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2

Step 1 Stride 8 — Thread IDs: 0 1 2 3 4 5 6 7

Values	8	-2	10	6	0	9	3	7	-2	-3	2	7	0	11	0	2

Step 2 Stride 4 — Thread IDs: 0 1 2 3

Values	8	7	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2

Step 3 Stride 2 — Thread IDs: 0 1

Values	21	20	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2

Step 4 Stride 1 — Thread IDs: 0

Values	41	20	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2

In each step, each thread in the block sums two values and writes them back in shared memory.

Between each step there should be a `__sync_threads()`

5

Sum reductions in CUDA

```
if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads();
if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads();
if (tid < 64 ) { sdata[tid] += sdata[tid + 64];  } __syncthreads();
// warp reduce
if (tid < 32) {
  sdata[tid] += sdata[tid + 32];
  sdata[tid] += sdata[tid + 16];
  sdata[tid] += sdata[tid + 8];
  sdata[tid] += sdata[tid + 4];
  sdata[tid] += sdata[tid + 2];
  sdata[tid] += sdata[tid + 1];
}
// result in sdata[0]
```

Threads in a warp don't need to be synchronized.
When only a warp operates, remove thread synchronization

Other interesting optimization we implemented: warp register shuffle instructions!
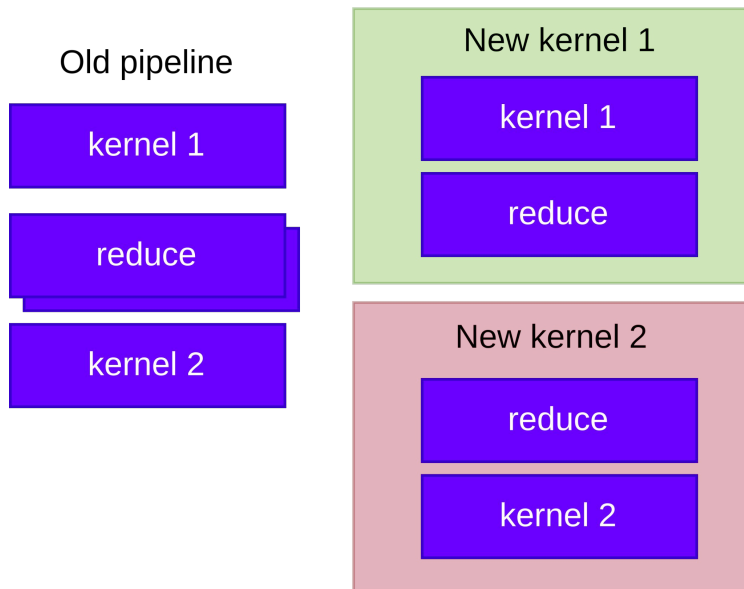(but we're short on time)

Optimizing reductions

Can we have less block-wise barriers?
Now we only do one final warp reduction, we can also use warp reductions before.

data (size: N)

warpReduce	warpReduce	warpReduce	warpReduce

partial_sum size: N / WARPS_PER_BLOCK (< WARP_SIZE)

- - - barrier!

warpReduce

out size: 1 (for each block)

Only 1 barrier!

Optimizing reductions

Old pipeline

kernel 1

reduce

kernel 2

New kernel 1

kernel 1

reduce

New kernel 2
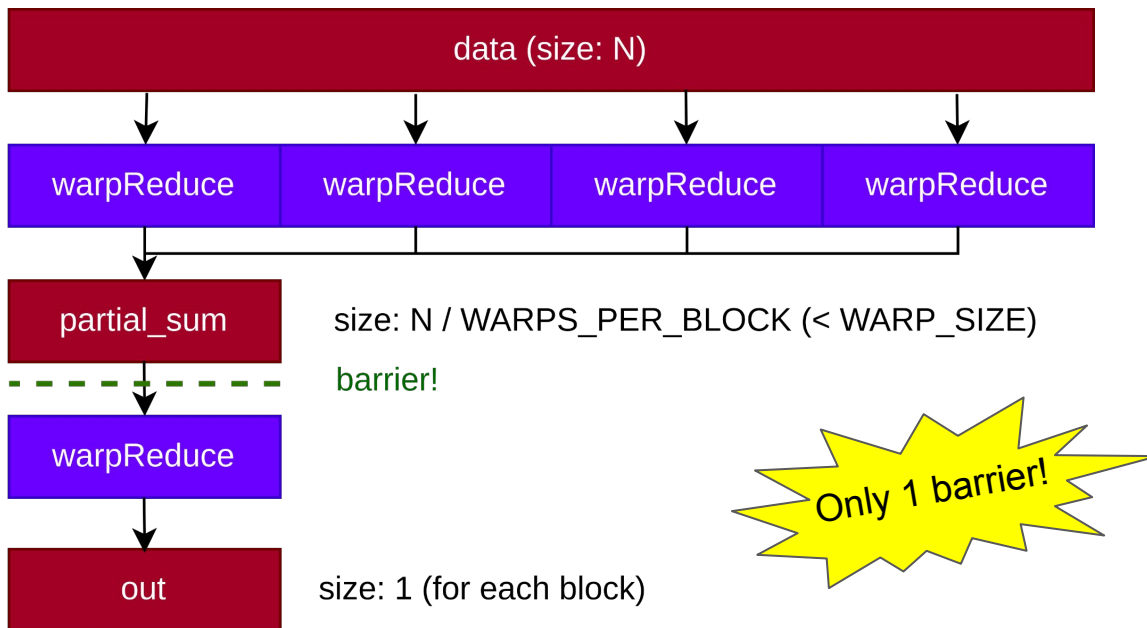
reduce

kernel 2

Only launch 2 kernels at each step

Also include sequential code in the kernels to remove memcopies

Halfing memory reads

New hotspot:

```
int j = k - i - 1;
DATA_TYPE yii = yi[i];
DATA_TYPE yij = yi[j];
yo[i] = yii + a*yij;
yo[j] = yij + a*yii;
```
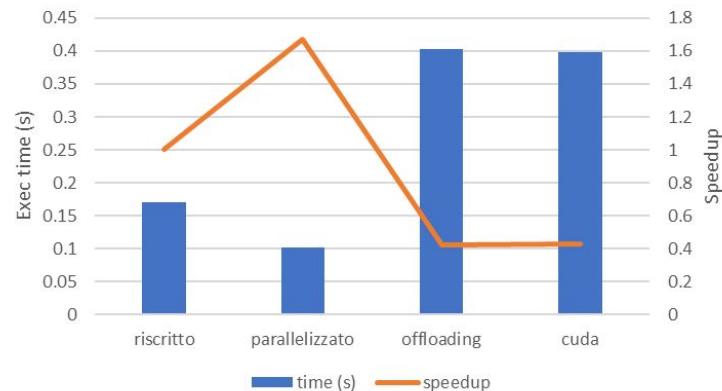
```
yo[i] = yi[i] + alpha * yi[k - i - 1];
```

`yo[i]` and `yo[k-i-1]` use the same memory locations.

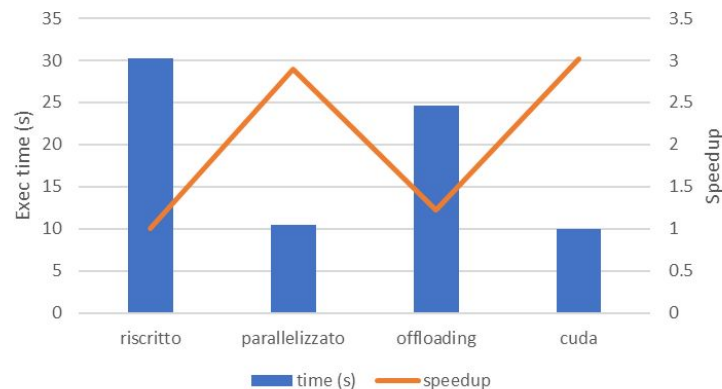We can compute both at the same time to save half of the global memory reads!
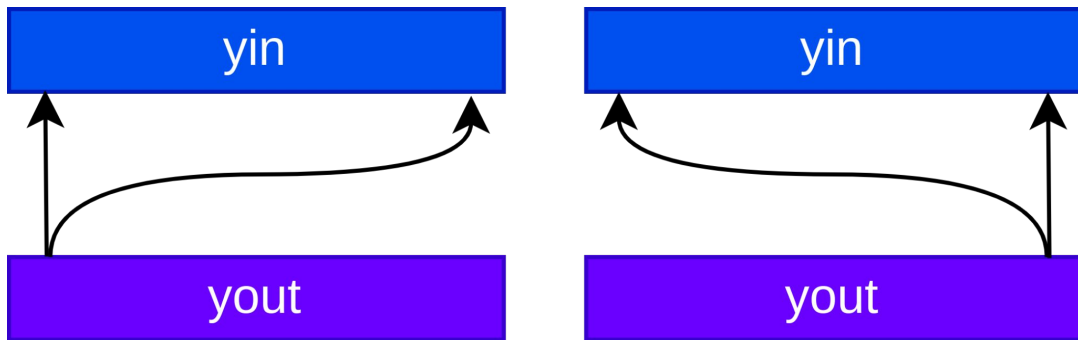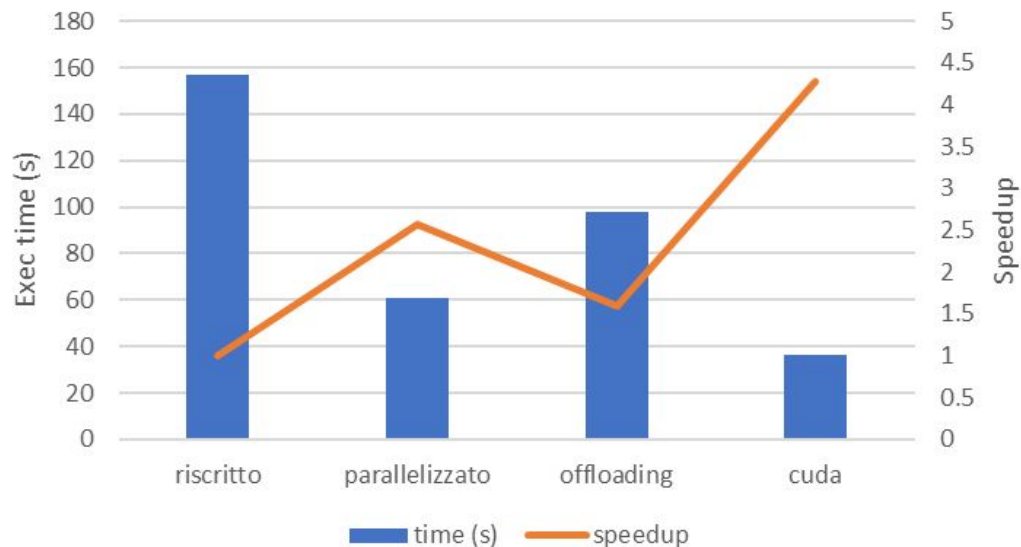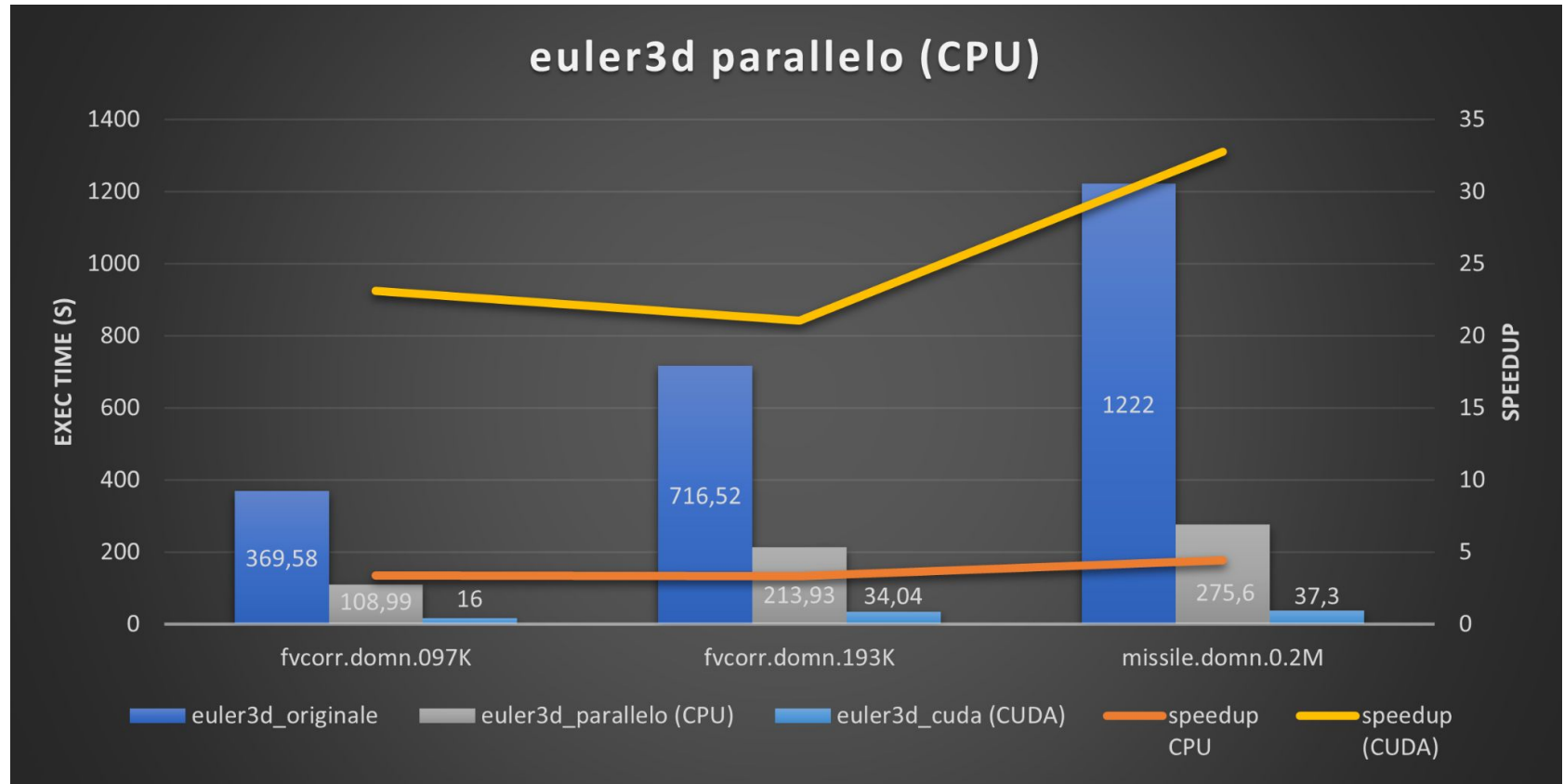
Durbin performance

Optional assignment

CONCLUSION

- GPU offloading introduces a lot of overhead (data transfer, extra-synchronization)

- While OpenMP can be easily used to parallelize already developed software, CUDA gives us fine-grained control over GPU hardware, this in many case is necessary to minimize overhead and maximize performance on GPU.

THANKS FOR YOUR ATTENTION!!