HPC - Assignment n°2

Durbin-Levinston algorithm

Salvatore Bianco, Linda Burchiellaro, Carlo Uguzzoni

Where did we leave off?

Working on our cuda kernel, we found out:

```
sum[0][k] = r[k];

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

This could be reduced!

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

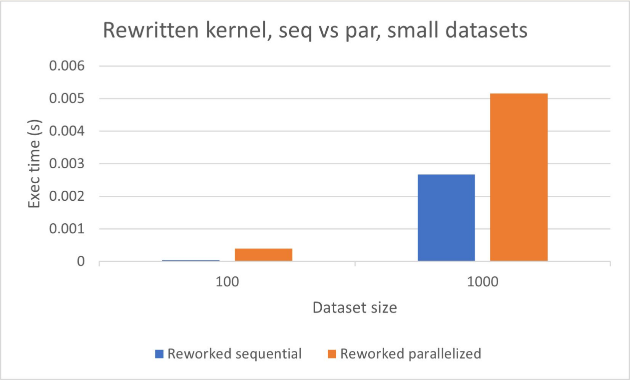
Also access pattern on matrix y could be changed!

So we obtained further speed-up!

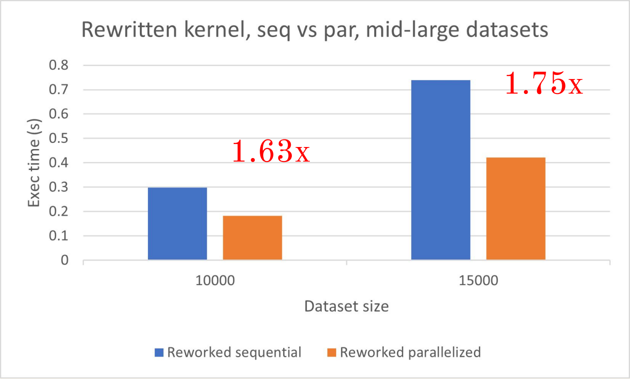
Version	Time (size = 100)	Time (size = 1000)	Time (size = 10000)	Time (size = 15000)
Sequential	0.00005	0.01169	4.41561	38.55401
Reworked sequential	0.00004	0.00267	0.29813	0.73846

About new changes and parallelization via openmp

No speedup at all for small datasets...



...but quite better for mid-large ones!



Expected speedup should be higher. We suspect this is happening because of thread awakening and synchronization, but it could also be a matter of bandwidth.

From base code to cuda kernel

```
  int i, k;
#pragma scop
  y[0][0] = r[0];
  beta[0] = 1;
  alpha[0] = r[0];

  for (k = 1; k < _PB_N; k++)
  {
    beta[k] = beta[k - 1] - alpha[k - 1] * alpha[k - 1] * beta[k - 1];
    sum[0][k] = r[k];

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

    alpha[k] = -sum[k][k] * beta[k];

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

    y[k][k] = alpha[k];
  }

  for (i = 0; i < _PB_N; i++)
    out[i] = y[i][_PB_N - 1];
}
```

- `alpha`, `beta`, `sum` to `__device__` variables
- only 2 1-d arrays for the two rows of `y` we actually work on
- only a minimal part of the old data structures to be allocated/initialized

Kernel 1 (sum reduction)

Kernel 2 (saxpy-like operation)

More in detail further on...

Deleted to spare a memcpy. Results are returned as the last row of `y` we computed.

Data structures cuda mallocs/memcpies

```
__device__ DATA_T d_alpha, d_beta, d_sum;
```

`__device__` variables helped removing sequential stages around the two kernels

```
// Device data structures.
DATA_T *d_r, *y_old, *y_new;

// Device mallocs.
gpuErrchk(cudaMalloc((void **)&d_r, sizeof(DATA_T) * N));
gpuErrchk(cudaMalloc((void **)&y_old, sizeof(DATA_T) * N));
gpuErrchk(cudaMalloc((void **)&y_new, sizeof(DATA_T) * N));
```

Just 3 linear arrays (size `n`) allocated

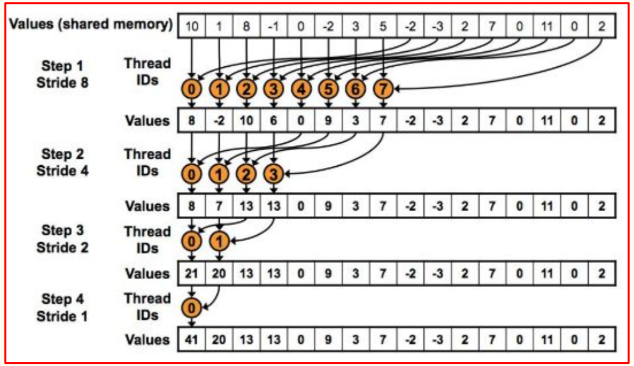
```
// Memcopies.
// Device's array r.
gpuErrchk(cudaMemcpy(d_r, h_r, sizeof(DATA_T) * N, cudaMemcpyHostToDevice));
// y_old[0] = r[0].
gpuErrchk(cudaMemcpy(y_old, d_r, sizeof(DATA_T), cudaMemcpyDeviceToDevice));
DATA_T alpha;
DATA_T beta = 1;
// alpha = r[0].
gpuErrchk(cudaMemcpy(&alpha, d_r, sizeof(DATA_T), cudaMemcpyDeviceToHost));
gpuErrchk(cudaMemcpyToSymbol(d_alpha, &alpha, sizeof(DATA_T)));
// beta = 1.
gpuErrchk(cudaMemcpyToSymbol(d_beta, &beta, sizeof(DATA_T)));

// Function kernel durbin call (device).
kernel_durbin_device(y_old, y_new, d_r);

// out = y_new dell'ultima iterazione di durbin (viene swappato -> quindi y_old).
gpuErrchk(cudaMemcpy(d_out, y_new, sizeof(DATA_T) * N, cudaMemcpyDeviceToHost));
```

- Minimum data motion from host to device
- `__device__` variables initialization (where needed)

Kernel 1 - sum reduction

```
// Device kernel.
// Variabili dislocate su device.
__device__ DATA_T d_alpha, d_beta, d_sum;

// Primo kernel -> calcolo del nuovo beta + calcolo delle somme parziali e successiva reduction su sum.
__global__ void first_kernel(DATA_T *__restrict__ y, DATA_T *__restrict__ r, int k)
{
  __shared__ DATA_T partialSum[BLOCK_SIZE];

  // Coordinate del thread.
  int tid = threadIdx.x;
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  // Calcolo del valore base di sum.
  d_sum = r[k];

  // Calcolo del nuovo beta.
  DATA_T beta = d_beta;
  d_beta = beta - d_alpha * d_alpha * beta;

  // Caricamento delle somme parziali in memoria condivisa.
  if (i < k)
    partialSum[tid] = r[k - i - 1] * y[i];
  else
    partialSum[tid] = 0;

  __syncthreads();

  // Riduzione. Ciascun blocco porta la propria somma parziale in partialSum[0].
  for (int stride = blockDim.x / 2; stride > 0; stride >>= 1)
  {
    if (tid < stride)
      partialSum[tid] += partialSum[tid + stride];

    __syncthreads();
  }

  // Il thread con tid 0 di ciascun blocco aggiorna il valore globale nel device con una atomicAdd.
  if (tid == 0)
    atomicAdd(&d_sum, partialSum[0]);
}
```

- Use of shared memory for best bandwidth
- Partial sums are stored in an array of size `BLOCK_SIZE`
- for loop operates a first sum reduction for each block on array index 0
- Every thread with `tid = 0` adds it's block partial sum to `__device__ d_sum` with an `atomicAdd`



Kernel 2 - saxpy

```
// Secondo kernel -> calcolo del nuovo alpha + calcolo del nuovo y in stile saxpy.
__global__ void second_kernel(DATA_T *__restrict__ y_old, DATA_T *__restrict__ y_new, int k)
{
  // Coordinate del thread.
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  // Calcolo del nuovo alpha.
  d_alpha = -d_sum * d_beta;

  if (i < k)
    y_new[i] = y_old[i] + d_alpha * y_old[k - i - 1];

  y_new[k] = d_alpha;

}
```

Basic saxpy-like kernel

```
// Funzione chiamante dei kernel. Replica il kernel di Durbin.
void kernel_durbin_device(
  DATA_T *__restrict__ y_old,
  DATA_T *__restrict__ y_new,
  DATA_T *__restrict__ d_r)
{
  int k;
  // int GRID_SIZE = (N + BLOCK_SIZE - 1) / BLOCK_SIZE;
  int GRID_SIZE;

  for (k = 1; k < N; k++)
  {
    GRID_SIZE = (k + BLOCK_SIZE - 1) / BLOCK_SIZE;

    // Calcolo del nuovo beta e di sum.
    first_kernel<<<GRID_SIZE, BLOCK_SIZE>>>(y_old, d_r, k);

    // Calcolo del nuovo alpha e del nuovo y.
    second_kernel<<<GRID_SIZE, BLOCK_SIZE>>>(y_old, y_new, k);

    // Scambio degli y.
    swapPointers(y_old, y_new);
  }
}
```

Grid size is determined by variable `k` for both kernel calls, to avoid creating non-working blocks

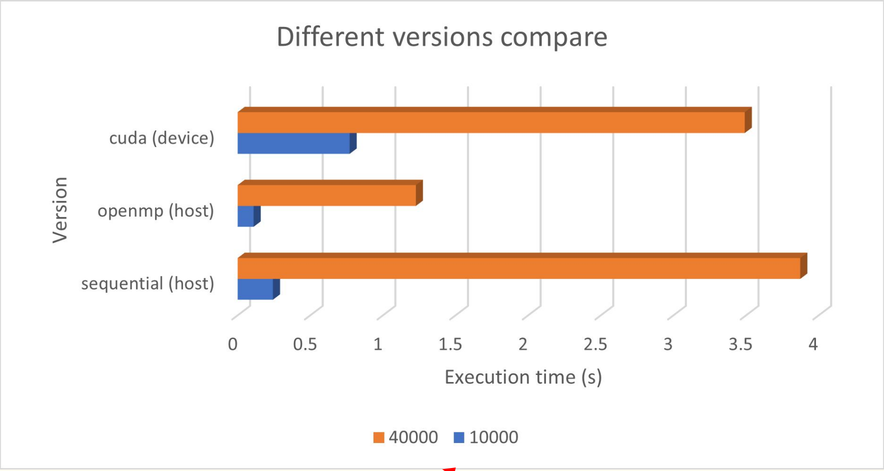
Results (overview)

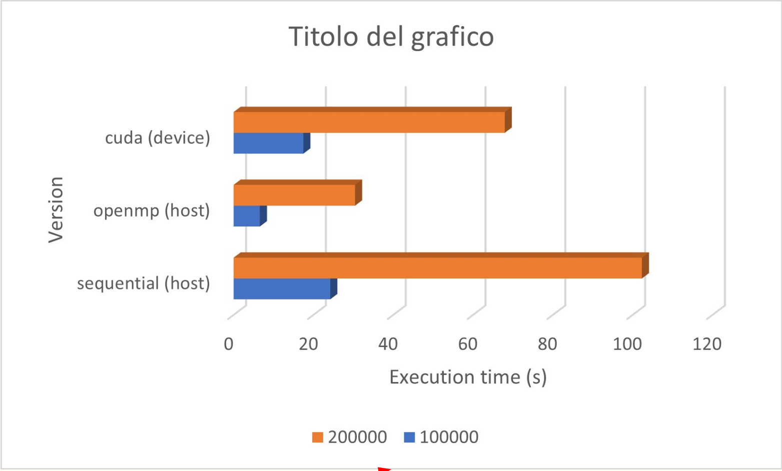
```
Durbin (Host) :     0.243 sec    8242.6 GFLOPS
Durbin (GPU):     0.762 sec    2624.8 GFLOPS
==13349== Profiling application: ./durbin_vfinal.exe
==13349== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   72.36%  120.75ms      9999  12.075us  2.3430us  22.868us  first_kernel(float*, float*, int)
                   27.63%  46.103ms      9999  4.6100us  1.0930us  7.9180us  second_kernel(float*, float*, int)
                    0.00%  7.5530us         3  2.5170us     209ns  7.0840us  [CUDA memcpy HtoD]
                    0.00%  4.8970us         2  2.4480us     834ns  4.0630us  [CUDA memcpy DtoH]
                    0.00%  2.3440us         1  2.3440us  2.3440us  2.3440us  [CUDA memcpy DtoD]
      API calls:   69.27%  725.53ms     19998  36.280us  34.167us  895.07us  cudaLaunchKernel
                   30.63%  320.76ms         3  106.92ms  18.646us  320.72ms  cudaMalloc
                    0.06%  588.60us         4  147.15us  97.919us  173.13us  cudaMemcpy
                    0.02%  249.07us         3  83.022us  21.667us  180.00us  cudaFree
                    0.01%  122.97us        97  1.2670us     677ns  31.772us  cuDeviceGetAttribute
                    0.01%  80.783us         2  40.391us  35.105us  45.678us  cudaMemcpyToSymbol
                    0.00%  10.052us         1  10.052us  10.052us  10.052us  cuDeviceTotalMem
                    0.00%  5.9880us         3  1.9960us  1.5100us  2.9160us  cuDeviceGetCount
                    0.00%  2.5000us         1  2.5000us  2.5000us  2.5000us  cuDeviceGetName
                    0.00%  2.4470us         2  1.2230us     989ns  1.4580us  cuDeviceGet
                    0.00%     937ns         1     937ns     937ns     937ns  cuDeviceGetUuid
```

Generic profile test executed on a 10k-sized dataset

- almost 100% computation on GPU happens inside the two kernels
  - Also the kernels sum up to ~160ms (which is faster than openmp parallelized version)
- almost 100% of API calls happens in:
  - `cudaMalloc` (unavoidable)
  - `cudaLaunchKernel`: 2 calls per iteration in the outermost `for` loop, which carries dependencies

Results (charts)





Dataset sizes

Possible further improvements

- First kernel could be improved:
  - `atomicAdd` is sub-optimal. Could write a new implementation which doesn't use it, thus 100% based on block reductions
  - By also applying reductions on single warps instead of blocks (they need no synchronization)
- Second kernel could be improved:
  - Memory reads happen twice on the same array `y` elements (`y_old[i]` and `y_old[k - i - 1]`). Surely there's some way to halfen memory accesses

Conclusions

With all summed up, this cuda version is still faster than the original version of the kernel, also it tends to get faster than the rewritten kernel on very large datasets. Yet the `openmp` version is always faster, due to the lack of overhead.