High Performance Computing

Parallelize dynprog.c
using CUDA

A project made by Francesco Malferrari, Gianluca Siligardi and Andrea Somenzi

We start from rewritten code

dynprog.c

```c
/* Main computational kernel. The whole function will be timed,
   including the call and return. */
static void kernel_dynprog(int tsteps, int length,
                           DATA_TYPE POLYBENCH_2D(c, LENGTH, LENGTH, length, length),
                           DATA_TYPE POLYBENCH_2D(W, LENGTH, LENGTH, length, length),
                           DATA_TYPE POLYBENCH_3D(sum_c, LENGTH, LENGTH, LENGTH, length, length, length),
                           DATA_TYPE *out)
{
  int iter, i, j, k;

  DATA_TYPE out_l = 0;

  for (iter = 0; iter < _PB_TSTEPS; iter++)
  {
    for (i = 0; i <= _PB_LENGTH - 1; i++)
      for (j = 0; j <= _PB_LENGTH - 1; j++)
        c[i][j] = 0;
    for (i = 0; i <= _PB_LENGTH - 2; i++)
    {
      for (j = i + 1; j <= _PB_LENGTH - 1; j++)
      {
        sum_c[i][j][i] = 0;
        for (k = i + 1; k <= j - 1; k++)
          sum_c[i][j][k] = sum_c[i][j][k - 1] + c[i][k] + c[k][j];
        c[i][j] = sum_c[i][j][j - 1] + W[i][j];
      }
    }
    out_l += c[0][_PB_LENGTH - 1];
  }
  *out = out_l;
}
```

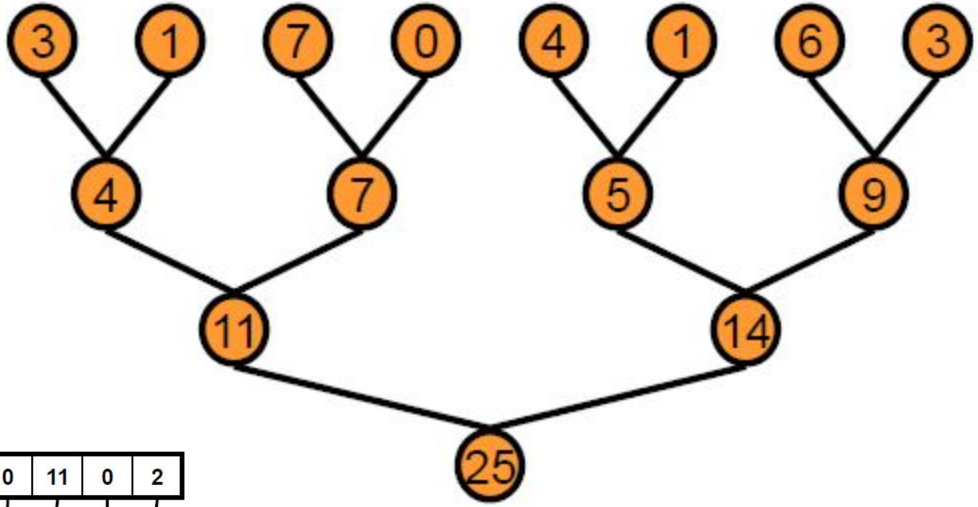
rew_dynprog.c

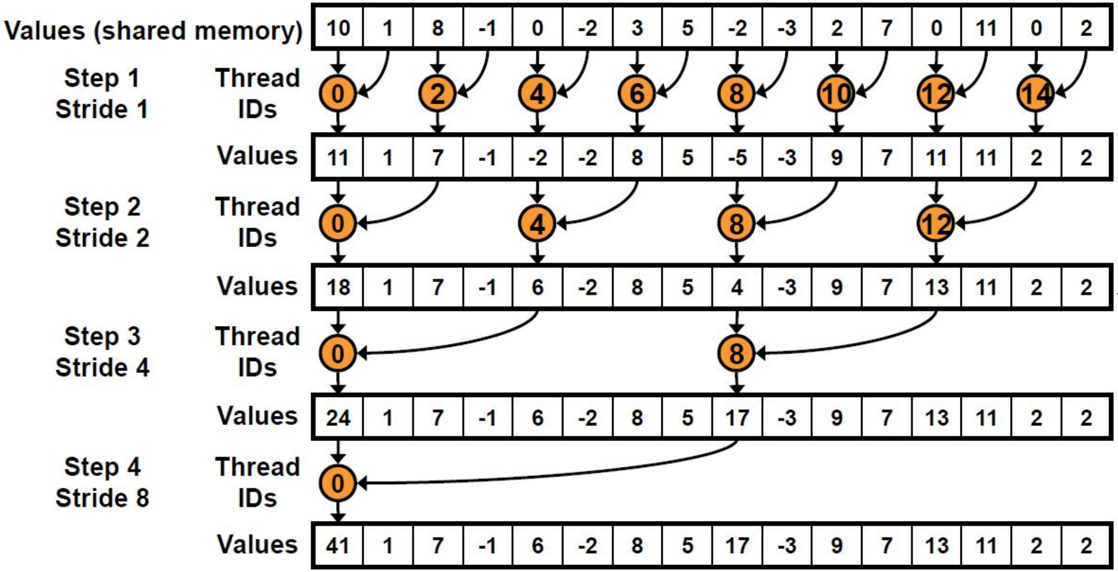
```c
/* Main computational kernel. The whole function will be timed,
   including the call and return. */
static void kernel_dynprog(int tsteps, int length,
                           DATA_TYPE POLYBENCH_1D(c, LENGTH, length),
                           DATA_TYPE POLYBENCH_1D(W, LENGTH, length),
                           DATA_TYPE sum_c,
                           DATA_TYPE *out)
{

  DATA_TYPE out_l = 0;
  sum_c = 0;

  for (int i = 1; i < _PB_LENGTH; i++)
  {
    for (int j = 1; j < i; j++)
      sum_c += c[j];
    c[i] = sum_c + W[i];
    sum_c = 0;
  }

  for (int k = 0; k < _PB_TSTEPS; k++)
    out_l += c[_PB_LENGTH - 1];

  *out = out_l;

}
```

<u>We didn't reinvent the wheel</u>

CUDA reduction

- Combine multiple blocks in one
- Multiple kernels
- Interleaved addressing






```
for (unsigned int s=1; s < blockDim.x; s *= 2)
{
        if (tid % (2*s) == 0) {
                sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
}
```

Images source: [1]

The implementation (1)

```
__global__ void partial_sum(DATA_TYPE *input, DATA_TYPE *output, int size) {
  extern __shared__ double sharedData[];

  int tid = threadIdx.x;
  int gid = blockIdx.x*(blockDim.x*2) + threadIdx.x;

  if (gid < size) {
    sharedData[tid] = input[gid] + input[gid+blockDim.x];
  }
  else {
    sharedData[tid] = 0;
  }

  __syncthreads();

  for (int offset = blockDim.x / 2; offset > 32; offset >>= 1) {
    if (tid < offset) {
      sharedData[tid] += sharedData[tid + offset];
    }
    __syncthreads();
  }

  if (tid < 32)
    warpReduce(sharedData, tid);

  if (tid == 0)
    output[blockIdx.x] = sharedData[0];
}
```

```
__device__ void warpReduce(volatile double* sdata, int tid) {
  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];
}
```

The implementation (2)

```
__global__ void update_c(DATA_TYPE *c, DATA_TYPE *part_sum, DATA_TYPE *W, int i)
{
  int thid = threadIdx.x;

  if(thid == 0) {
    c[i] = part_sum[0] + W[i];
  }
}
```

```
for(int i = 1; i <= length; i++)
{

  if (i < BLOCK_SIZE)
  {
  int is_numBlocks = (i + blockSize - 1) / blockSize;
      partial_sum<<<is_numBlocks, blockSize, blockSize * sizeof(DATA_TYPE)>>>(d_c, d_out_l, i);
      update_c<<<1, 1>>>(d_c, d_out_l, d_W, i);
  }
  else if (i < BLOCK_SIZE * BLOCK_SIZE)
  {
  int is_numBlocks = (i + blockSize - 1) / blockSize;
      partial_sum<<<is_numBlocks, blockSize, blockSize * sizeof(DATA_TYPE)>>>(d_c, d_pout_l, i);
  int is_numBlocks2 = (is_numBlocks + blockSize - 1) / blockSize;
  partial_sum<<<is_numBlocks2, blockSize, blockSize * sizeof(DATA_TYPE)>>>(d_pout_l, d_out_l, i);
  update_c<<<1, 1>>>(d_c, d_out_l, d_W, i);
  }
  else
  {
  int is_numBlocks = (i + blockSize - 1) / blockSize;
              partial_sum<<<is_numBlocks, blockSize, blockSize * sizeof(DATA_TYPE)>>>(d_c, d_pout2_l, i);
              int is_numBlocks2 = (is_numBlocks + blockSize - 1) / blockSize;
              partial_sum<<<is_numBlocks2, blockSize, blockSize * sizeof(DATA_TYPE)>>>(d_pout2_l, d_pout_l, i);
  int is_numBlocks3 = (is_numBlocks2 + blockSize - 1) / blockSize;
              partial_sum<<<is_numBlocks3, blockSize, blockSize * sizeof(DATA_TYPE)>>>(d_pout_l, d_out_l, i);
              update_c<<<1, 1>>>(d_c, d_out_l, d_W, i);
  }

}
cudaDeviceSynchronize();
```

Parallel Prefix Sum (Scan) with CUDA

Scan: out[k] := in[k-1] + out[k-1]

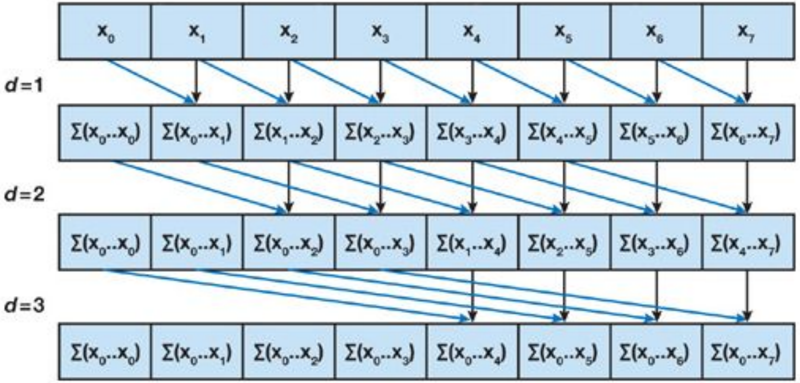
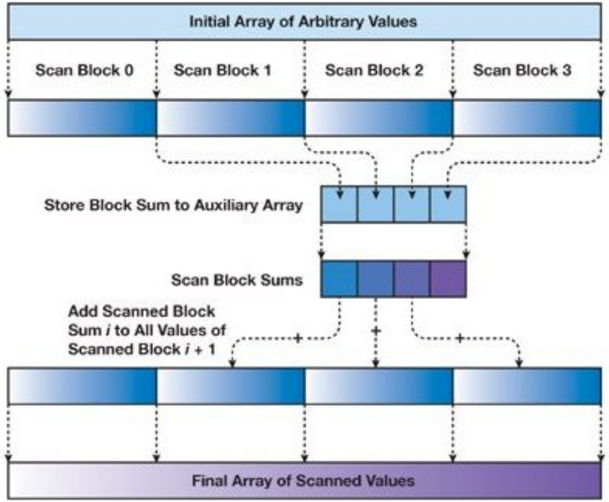


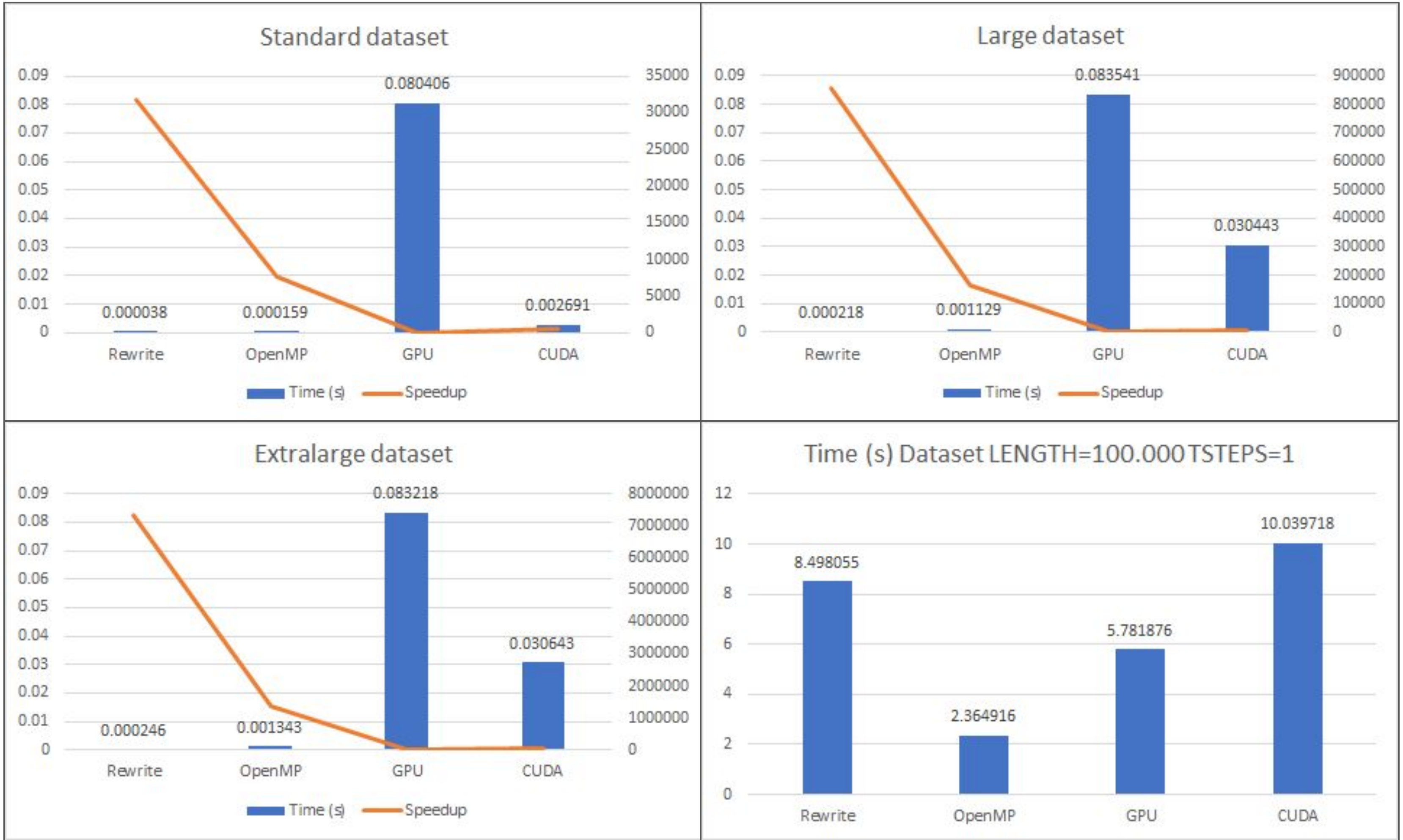
Figure 39-2 The Naive Scan of



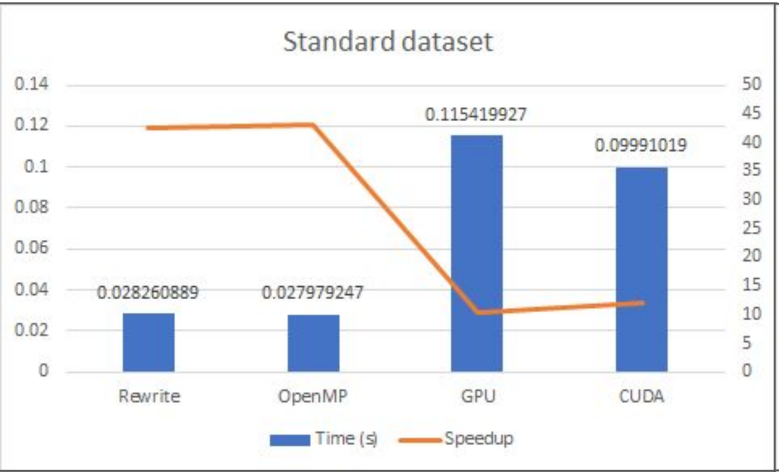
Hillis and Steele assumes that there are as many processors as data elements. For large arrays on a GPU running CUDA, this is not usually the case.

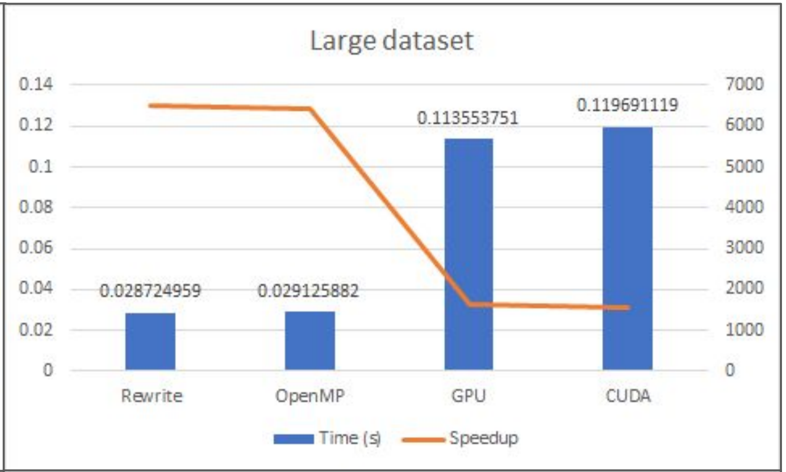
Images source: [2]

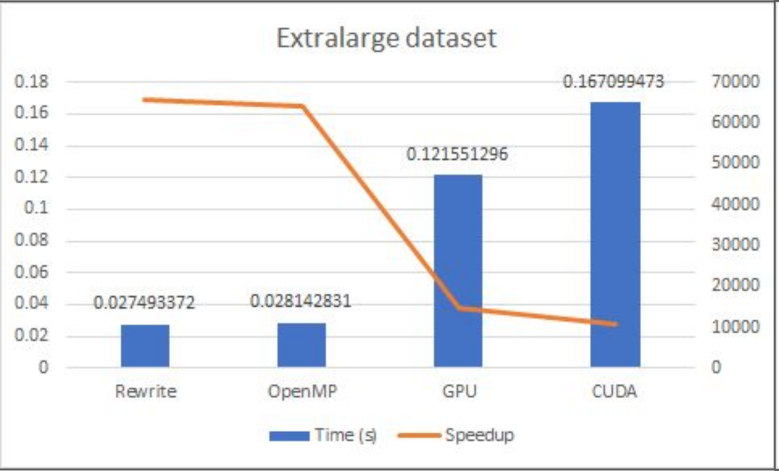
Final comparison performances (using polybench time)

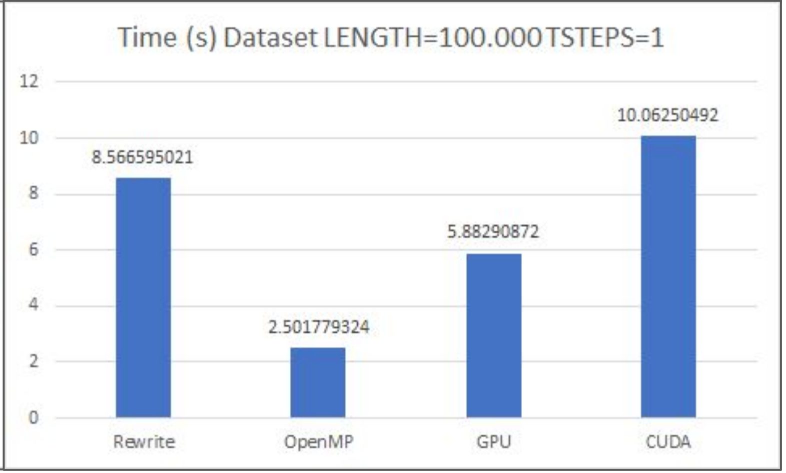


Final comparison performances (using perf)









Further improvements?

- Thrust or Cub Cuda Libraries
- But out of scope

Thanks for the attention

[1]: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf
[2]: *Chapter 39. Parallel Prefix Sum (Scan) with CUDA*. (n.d.). NVIDIA Developer. https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda