PROGRAMACIÓN PARALELA
CUDA Tutorial

In the accompanying archive, there is a compressed directory called `Lab08`. Once the archive is uncompressed change to that directory. Inside it there are three directories: `03-AnyVectorAdd`; `04-DotProduct`; `05-dist`.

The `03-AnyVectorAdd` directory has three programs: `vectorAddition.c`; `04-DotProduct`; `05-dist`.

The `vectorAddition.c` program just adds two vectors (`augend` and `addend`) of length `N` using the CPU. It follows the same pattern seen in the previous version of the program, but now the computation is verified by comparing the results obtained by adding the two vectors using the `for` loop inside the `add` function against the product `2 * i` (our golden reference) for each entry in the resulting vector (`result`).

The `vectorAddition_blocks_threads.cu` program does the same job, but in this case, the vector addition is done in the GPU. According to the kernel launching configuration parameters, the `add` kernel will be executed using `BLOCKS` blocks and `THREADS_PER_BLOCK` threads per block for a total of `BLOCKS` times `THREADS_PER_BLOCK` threads. Each of these threads will execute the `add` kernel.

Therefore, there is no need for a `for` loop inside the kernel. The *Thread Block Scheduler* (*TBS*) is a hardware scheduler on the GPU that dispatches CUDA kernel's thread blocks to the GPU's *Streaming Multiprocessors* (*SMs*).

Since the number of elements to process, `N`, is bigger than the number of threads we will use to operate upon them, we need to make several iterations. This is precisely the purpose of using the `while ( index < N )` statement in the add kernel. The `index += BLOCKS * THREADS_PER_BLOCK` statement adjust the `index` variable so that the next iteration continues one stride ahead of the previous one. Besides, the conditional expression makes sure that we will not write outside the allocated memory.

The `vectorAddition_blocks_threads_t.cu` program just adds event management functions of the CUDA runtime API to measure elapsed execution times.

For comparison purposes, the `vectorAddition_t.c` program does an equivalent elapsed time measurement.

Besides, the `vectorAddition_scaleblocks_threads_t.cu` uses a multiple of the number of SMs available in the GPU as determined by using the `cudaGetDeviceProperties` function.

The `04-DotProduct` directory has three programs: `dotp.c`; `dotp_blocks_threads.cu`; `dotp_v1.cu`; `dotp_v2.cu`.
The `dotp.c` program calculates the dot product of two vectors (`v1` and `v2`) of length `N` using the CPU. The dot product is stored in the `result` variable.
Since we know that

$$\sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

the code is verified by comparing its computed result against evaluating the right-hand of the above equation (our golden reference).

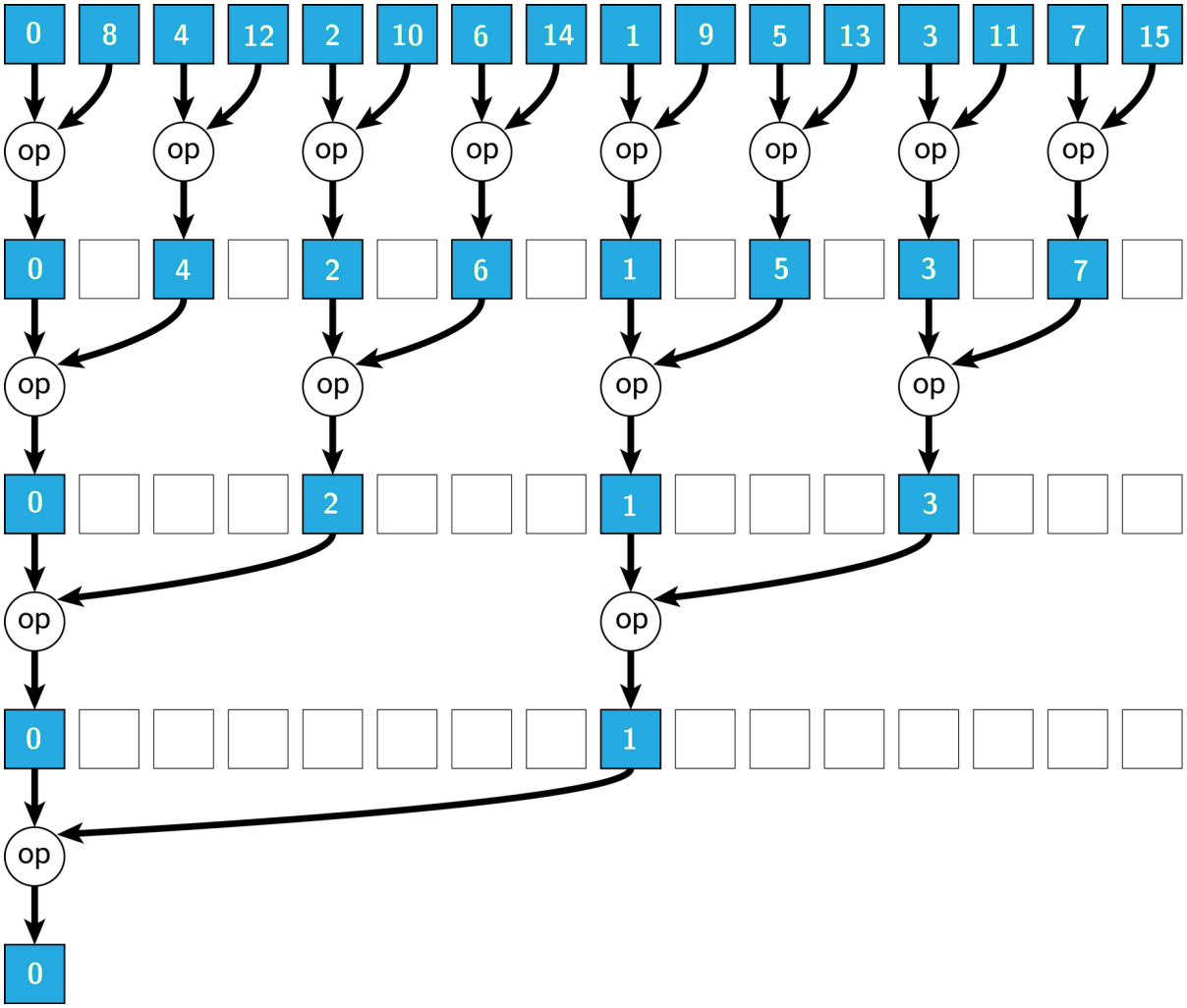
The `dotp_blocks_threads.cu` calculates the dot product of two vectors, `v1` and `v2`; the `mult_vects` kernel calculates the product of pairwise elements and is stored in a third vector, `vy`. The contents of `vy` is added using the CPU.

The `dotp_v1.cu` uses the shared memory inside the GPU. Because it is on-chip, shared memory is much faster than local (GPU card memory) and global (system) memory. In fact, shared memory latency is roughly 100x lower than uncached global memory latency.

The `shm_bdp[THREADS_PER_BLOCK]` vector is allocated per block, so all threads in every block have access to a private copy of this vector. The `tdp` variable stores the dot product of the sub-vectors processed by a single thread. The `shm_bdp[threadIdx.x]` entry, then stores the dot product of the sub-vectors processed by a single thread.

When sharing data between threads, race conditions may arise. Hence, to ensure correct results when parallel threads cooperate, we must synchronise the threads to avoid RAW hazards. That is precisely the purpose of using the CUDA `__syncthreads` function; in fact, the function provides a barrier mechanism for all threads in a block.

Then, we need to reduce the partial per-thread dot products. Here we apply a parallel reduction algorithm. It repeatedly groups the array arguments in pairs; each pair is computed in parallel with others, halving the overall array size in one step. The process is repeated until only a single element exists as shown in the below Figure for 16 threads. Each blue box represent a thread, whose id is denoted by the white number inside the blue box. The `op` inside the white bubble represents the reduction operation. Each row of threads represents an iteration across the entire reduction. The number of iterations is given by $Log_2(t)$, where t is the number of threads. As can be seen, the approach is the same used before when dealing with non-collective MPI reductions.



Finally, thread 0 in each block writes the per-block dot product in the `vyb[blockIdx.x]` entry. The `vyb` vector has `BLOCKS` entries, which are added on the CPU.

The `dotp_v2.cu` also uses the shared memory inside the GPU. In this case, the `shm_tdp[THREADS_PER_BLOCK]` vector is allocated per block, so all threads in every block have access to a private copy of this vector. It stores the dot product of the sub-vectors processed by a single thread. Each `shm_tdp[threadIdx.x]` entry, then stores the dot product of the sub-vectors processed by a single thread.

Once all the threads in a block have reached the barrier, thread 0 accumulates the per-thread dot product for all the threads in the block in the `bdp` variable.

Finally, the per-thread dot products are reduced using the CUDA provided `atomicAdd` function.

The `05-dist` directory has two programs: `dist.c`; `dist.cu`.

The `dist.c` program computes on the CPU an array of distances (`out`) from a reference point (`ref`) to each of `N` points uniformly spaced along a line segment.

The `dist.cu` program does the same job on the GPU. It then illustrates how to declare functions (`scale` and `distance`) that will be executed on the GPU and called from the CUDA kernel (`distanceKernel`).