Parallel programming on GPUs

Briefing about NVIDIA architectures, CUDA, OpenACC, OpenCL and HSA

Francesco Ostidich

July 16, 2024

Architecture

GPUs take advantage of superscalar architectures: each core contains several ALUs that are able to execute the same instruction of many thread in parallel.

This approach, called SIMT (single instruction multiple threads), resembles the SIMD execution model.

Many cores and other functional units are packed into a streaming multiprocessor; each streaming multiprocessor receives blocks of threads to be executed.

Furthermore, blocks are organized in a grid.

NVIDIA architectures

1. Tesla
2. Fermi: the number of cores has been increased, along with a higher precision. Address space has been unified, now having a shared memory per SM with an L1/L2 caches for global DRAM accesses. RF can manage 48 warp contexts.
3. Kepler: the new SMX can exploit ILP executing two instructions per warp. Dynamic parallelism has also been introduced. Furthermore, single and double precision cores have been separated, with the addition of texture filtering units. The memory hierarchy now contains a read-only partition connected to the L2 cache.
4. Maxwell: the SMM uses four schedulers, each assigned with a set of cores. Texture memory is also introduced.
5. Pascal: the unified virtual address space permits a transparent transmission between host and device. NVLink allows multi-GPU systems. Single, double and half precision cores have been separated.
6. Volta: multiple applications can execute concurrently on a single GPU. Streaming multicores are partitioned in processing blocks; floating point and integer cores have been separated, along with the introduction of tensor cores specialized for matrix multiplication.
7. Turing: introduction of ray tracing units.
8. Ampere: GPUs are multi-instance (the device can be seen as seven separate distinct GPUs). Tensor cores have been perfected.

Graphics pipeline

GPUs real-time 3D rendering is achieved by executing a series of steps.

Firstly, the CPU assembles a vertex model from a descriptor stored in memory; the GPU then renders the full object starting from the geometry information.

1. Vertex processing: after pulling geometry information from memory, vertexes from object space, received from the CPU, are converted to screen clip space.
2. Primitives setup: vertexes are converted into triangles; perspective and viewpoint are applied, and a clip is executed on backfacing and out-of-view triangles.
3. Fragment rasterization: screen space geometry is rasterized to pixels, by determining which primitive they overlap with; fragments are assigned a final color (texture mapping, light shaders, anti-aliasing).
4. Memory interface: pixels are copied into the frame buffer of the screen.

CUDA Program structure

Parallelization can be accomplished by providing the GPU with kernels (functions) and the corresponding vectorized data.

Serial code is executed on the host (CPU), parallel code on the device (GPU).

When calling a kernel on the GPU, the structure of the program contains:

1. device memory allocation;
2. host to device data transmission;
3. kernel launch;
4. device to host data transmission;
5. device memory freeing.

When writing a kernel some restrictions are applied. For example the return type must be void, and the device memory is the only accessible.

Each function is labelled with qualifiers to define where it will be executed and who can call it.

- `__device__`: executed on the device and callable from the device
- `__global__`: executed on the device and callable from the host
- `__host__`: executed on the host and callable from the host

Kernel launch

The GPU will spawn several threads, that are grouped in blocks, themselves organized in a grid.

`cudaMalloc` is used to allocate memory on the device memory.

Data is transferred from CPU to GPU with `cudaMemcpy`, specifying the direction with `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`, `cudaMemcpyHostToHost`.

The kernel can be launched after setting the block and grid dimensions with the `dim3` struct (that has 3 dimensions).

In the end, device memory can be freed with `cudaFree`.

```
// N is the number of threads

cudaMalloc(&dv, N * sizeof(int));
cudaMemcpy(dv, hv, N * sizeof(int), cudaMemcpyHostToDevice);

kernelFunction1D<<<N / 256, 256>>>(dv);

dim3 blocksPerGrid(XLEN / 32, YLEN / 32, 1);
dim3 threadsPerBlock(32, 32, 1);
kernelFunction2D<<<blocksPerGrid, threadsPerBlock>>>(dv);

cudaMemcpy(hv, dv, N * sizeof(int), cudaMemcpyDeviceToHost);
cudaFree(dv);
```

Since every CUDA function returns a flag it is easy to do error handling. For example a `CHECK` MACRO can be introduced to be used with every call.

```
#define CHECK(call) { \
    const cudaError_t err = call; \
    if (err != cudaSuccess) { \
        printf("Error %s in file %s at line %d\n", \
            cudaGetErrorString(err), __FILE__, __LINE__); \
        exit(EXIT_FAILURE); \
    } \
}
```

Since kernels return void, their execution can only be checked with `cudaGetLastError`.

CUDA provides the API to query GPU information. By running

```
int dev;
cudaDeviceProp devProp;
cudaGetDevice(&dev);
cudaGetDeviceProperties(&devProp, dev);
```

the `devProp` struct will collect data like `major`, `minor`, `name`, `totalGlobMem`, `regsPerBlock`, `maxThreadsPerBlock`, `maxThreadsDim[3]`.

Data decomposition

Since C and CUDA only allow 1D vector allocations, multidimensional data has to be accessed by computing the index of the linearized vector.

Some CUDA built-in struct variables are used to identify the index of the current thread. All these have 3 dimensions (x, y, z).

- `blockIdx`: ID of the block in the grid
- `blockDim`: size of the block (number of threads)
- `threadIdx`: ID of the thread in the block
- `gridDim`: size of the grid (number of blocks)

```
// Host
cudaMalloc(&dv, sizeof(int) * NCOLS * NROWS);

// Device
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
dv[j * NCOLS + i]++;
```

Execution model

Each block is assigned to an SM by the global scheduler. An SM can have assigned multiple active blocks.

Inside the SM the block is divided in warps (32 threads each) that are interleaved during execution; warps can be stalled, eligible or selected.

Block size should be multiple of the warp size, but if that's not possible, dimensions checks are to be added. In this case the last threads of the last block will do nothing.

```
// Host
kernelFunction1D<<<ceil(N / 256.0), 256>>>(dv);

// Device
int i = blockIdx.x * blockDim.x + threadIdx.x;
if (i < N) dv[i]++;
```

Maximizing SM occupancy is key to optimal performances. The block size has to be tuned by means of a systematic profiling.

Some guidelines may be to avoid small blocks and choose a number of blocks larger than the number of SMs.

To compute the occupancy of a kernel it is possible to do as following.

```
cudaGetDeviceProperties(&prop, device);
cudaOccupancyMaxActiveBlocksPerMultiprocessor(
    &numBlocks, kernelFunction, blockSize, 0);
int activeWarps = numBlocks * blockSize / prop.warpSize;
int maxWarps = prop.maxThreadsPerMultiProcessor / prop.warpSize;
double occupancy = activeWarps / maxWarps;
```

It is also possible to compute the potential block size directly in the source code.

```
cudaOccupancyMaxPotentialBlockSize(
    &minGridSize, &blockSize, (void *) kernelFunction, 0, N);
int gridSize = (N + blockSize - 1) / blockSize;
kernelFunction<<<gridSize, blockSize>>>(dv);
```

Another thing to keep in mind is to avoid warp divergence. Data has to be partitioned so that all threads in the same warp take the same direction when encountering branches.

Synchronization

CUDA calls may be blocking or non-blocking based on whether the host code waits the function returns (e.g. memory copies) or if it continues asynchronously (e.g. kernel launch).

Barriers like `cudaDeviceSynchronize` can be used to force a wait.

Threads can be synchronized also within the same block with `__syncthreads`. Note that there's no barrier call available among different blocks; in this case the kernel must be split in two.

Kernel execution time can be measured either with a host-based counter or with CUDA events on device.

```
cudaEvent_t start, stop;
float time;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);
kernelFunction<<<N / 256, 256>>>(dv);
cudaEventRecord(stop);

cudaDeviceSynchronize();
cudaEventElapsedTime(&time, start, stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Memory access

The device contains various memory and cache banks organized hierarchically. Note that this arrangement can change between architectures.

A register file is placed at a core/SM level to contain the contexts of all threads. SMs also have a shared memory bank to be used by each block, along with a local memory partitioned between threads.

Constant and texture memories can be found at an SM/device level to store read-only data. Since global device memory transactions are slow, a L1 cache is available for each SM and a L2 cache for global usage.

The data transmissions between host and device can be executed autonomously by the CUDA framework via the usage of the `cudaMallocManaged` function. The memory allocated in this way can be accessed by both the host and the device transparently.

Global variables can be labelled with the `__device__` qualifier to be stored in the device global memory, and moved between host and device with `cudaMemcpyToSymbol` and `cudaMemcpyFromSymbol`.

```
__device__ int dv[N];
int hv[N];

// Host
cudaMemcpyToSymbol(dv, hv, N * sizeof(int));
kernelFunction<<<N / 256, 256>>>();
cudaMemcpyFromSymbol(hv, dv, N * sizeof(int));
```

In the same way, read-only data can be stored in the device constant memory by tagging the variable with `__constant__`; this memory is constant just for the device, but potentially modifiable by the host before the kernel launch.

```
__constant__ int dv[N];
int hv[N];

// Host
cudaMemcpyToSymbol(dv, hv, N * sizeof(int));
kernelFunction<<<N / 256, 256>>>();
cudaMemcpyFromSymbol(hv, dv, N * sizeof(int));
```

Texture memory (optimized for 2D data) can be accessed by using `const __restrict__` on the pointer parameter and forcing the read-only with `__ldg`. The data transfer made in the host is done in the usual way with `cudaMemcpy`.

```
__global__ kernelFunction(const unsigned char * __restrict__ v) {
    /*...*/
    unsigned char p = __ldg(v[i][j]);
    /*...*/
}
```

Variables can also be labelled with `__shared__` to explicitly allocate them in the shared memory. Allocation can be static (up to 3D arrays) or dynamic (only a 1D array).

```
// Static allocation
// Device
__shared__ int arr[N];
arr[threadIdx.x]++;
__syncthreads();
arr[N - threadIdx.x]--;

// Dynamic allocation
// Device
extern __shared__ int arr[];
// Host
kernelFunction<<<N / 256, 256, N * sizeof(int)>>>();
```

The global memory (slow) is accessed via transactions. A transaction is an aligned memory access to the same line/sector; accesses to sectors of the same line are packed in a single transaction. Data is aligned in blocks of 32 bytes.

Global memory usage is optimal when memory access is coalescent, achieved when 32 threads access a contiguous chunk of memory with the same transaction.

Unaligned requests require more than one transaction. Same data cannot be accessed by more than one thread at a time.

On global and shared memory it is possible to use atomic operations, i.e. functions that do not race to perform memory writes. Examples would be `atomicAdd`, `atomicMax` or `atomicExch`.

To avoid using memory to exchange data stored in registers, the shuffle instruction `__shfl` can share a value to all the threads in the warp.

Matrix multiplication

Let M and N be two square matrices with a dimension that is a multiple of the block size. The objective is to execute the matrix multiplication in order to obtain the final matrix P.

Each matrix is subdivided in tiles, that are the submatrices with dimension the block size. This is done to optimize memory access, since each tile corresponds to a single block. $M_{a,b}$ represents the tile of M in position a and b; $m_{c,d}$ represents the cell of M with c and d as coordinates.

To calculate the values in $P_{a,b}$ it is necessary to read all the data in the tile in row a of M and column b of N. However, this data is segmented between multiple blocks. For this reason the computation is executed in stages, so that each block accesses just one tile of M and one of N.

This workflow involves the usage of the shared memory to optimize matrix tile reading. In this way matrices in the shared memory are used multiple times to compute different tiles of the final matrix.

For example, the following code computes matrix multiplication for square matrices.

```
__global__ void sq_matrix_mul(int *M, int *N, int *P) {
    __shared__ int M_tile[BLOCK_WIDTH][BLOCK_WIDTH];
    __shared__ int N_tile[BLOCK_WIDTH][BLOCK_WIDTH];
    int bx = blockIdx.x, by = blockIdx.y;
    int tx = threadIdx.x, ty = threadIdx.y;
    int i = bx * BLOCK_WIDTH + tx;
    int j = by * BLOCK_WIDTH + ty;
    int p = 0;

    for (int b = 0; b < MATRIX_DIM / BLOCK_WIDTH; ++b) {
        M_tile[ty][tx] = M[j * MATRIX_DIM + b * BLOCK_WIDTH + tx];
        N_tile[ty][tx] = N[(b * BLOCK_WIDTH + ty) * MATRIX_DIM + i];
        __syncthreads();

        for (int t = 0; t < BLOCK_WIDTH; ++t)
            p += M_tile[ty][t] * N_tile[t][tx];
        __syncthreads();
        }
    P[j * MATRIX_DIM + i] = p;
}
```

Streams

CUDA supports only blocking (synchronous) data transfers on pageable memory, since GPUs cannot access it safely.

To allow asynchronous memory transfers, the host can allocate a (page-locked) pinned memory in which to copy the data from its paged-memory. The non-blocking transfer will then be executed from this pinned-memory. It is possible to store data directly into the pinned memory to avoid performing the copy, but excessive allocations might degrade host performance.

Pinned memory allocations and asynchronous data transfers can be performed with `cudaMallocHost`, `cudaMemcpyAsync`, `cudaFreeHost`.

A stream is a queue for memory transfers or kernel launches. There are two types of streams: the default NULL stream is implicitly used by the program, whereas a non-default stream must be explicitly declared.

When using a stream the host memory must be declared as pinned.

```
cudaStream_t s;
cudaStreamCreate(&s);
cudaMalloc(&dv, N);
cudaMallocHost(&hv, N);

cudaMemcpyAsync(dv, hv, cudaMemcpyHostToDevice, s);
kernelFunction<<<B, T, 0, s>>>(dv);
cudaMemcpyAsync(hv, dv, cudaMemcpyDeviceToHost, s);
cudaStreamSynchronize(s);

cudaFree(dv);
cudaFreeHost(hv);
cudaStreamDestroy(s);
```

It is possible to create a stream with flags with `cudaStreamCreateWithFlags`; an example flag would be `cudaStreamNonBlocking`.

Operations in the same stream are executed in order and do not overlap. Operations from different streams are unordered and can overlap.

Commands in the default stream cannot overlap with other streams (unless they are non-blocking).

Non-default streams kernels launched one after the other can overlap. If between two non-default streams a kernel is run with the default stream, then the three streams will be executed sequentially.

Memory copies can overlap too with the same concepts.

Events can be used for synchronization purpose.

```
// Host-device sync
kernelFunction1<<<B, T, 0, stream1>>>();
cudaEventRecord(event1);
cudaEventSynchronize(event1);

// Intra device sync
kernelFunction2<<<B, T, 0, stream2>>>();
cudaEventRecord(event2, stream2);
kernelFunction3<<<B, T, 0, stream3>>>();
cudaStreamEventWait(stream3, event2);
```

Since a single data transfer per direction (H2D, D2H) is allowed at any given time, tasks should be split and dispatched on various streams so that memory copies and kernel launches are executed in parallel.

Dynamic parallelism

A kernel can launch other kernels.

If the work of a kernel is a heterogeneous set of tasks, it is possible to launch child kernels that work on a different number of those tasks.

Parent kernel is guaranteed to end after child kernel ends.

In childs, local and shared memory become non visible.

Child grids launched within a thread block are executed sequentially, since they are pushed on the NULL stream. Concurrent execution is achievable via the use of multiple streams.

```
__global__ void parentKernel(int *start, int *end) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    for (int j = start[i]; j < end[i]; j++)
        childKernel<<<ceil((end[i] - start[i]) / 256.0), 256>>>(
            start[i], end[i]);
}
```

OpenACC

Parallel code can be written in C/C++ by using a set of compiler directives.

OpenACC execution model resembles the CUDA one: work is arranged between gangs, workers and vectors; these are similar to grids, blocks and warps.

Directives are specified with `#pragma`.

```
#pragma acc <directive> <clauses>
```

The `kernels` directive leaves the compiler to autonomously parallelize the code.

```
#pragma acc kernels
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        c[i * N + j] = a[i * N + j] + b[i * N + j];
```

The `parallel` directive forces the compiler to follow the programmer commands.

```
#pragma acc parallel
{
    #pragma acc loop
    for (int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
    #pragma acc loop
    for (int i = 0; i < M; i++)
        f[i] = d[i] + e[i];
}

#pragma acc parallel loop
for(int i = 0; i < N; i++)
    #pragma acc loop
    for(int j = 0; j < N; j++)
        c[i * N + j] = a[i * N + j] + b[i * N + j];
```

There are many clauses that can be used:

- `private(var)`: variables can be marked as thread-private;
- `collapse(n)`: nested loops can be collapsed to be parallelized together;
- `seq`: some nested loop can be excluded from parallelization;
- `reduction(op:var)`: reduction operations can be automatically performed;
- `worker`, `worker(n)`: it is possible to directly specify how to arrange the actors.

Like the parallelization process, also memory transfers can be automatically managed by the compiler, if no specifications are made. With the `data` clause the programmer can explicitly manage transfers.

```
#pragma acc data copyin(a[0:N], b[0:N]) copyout(c[0;N])
{
    #pragma acc kernels
    for (int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}
```

The clause `create` can be used to allocate device memory.

To achieve asynchronous execution it is possible to use `async` and `wait`.

```
#pragma acc kernels async(n)
// This region is assigned with the n id

#pragma acc wait(n)
// The end of the region with the n id is waited
```

OpenCL

OpenCL's work is subdivided in work-items arranged in work-groups, that themselves form a grid. This organization resembles the CUDA one.

Given a 1-dimensional problem, the kernel dimensions are obtainable with

- `get_work_dim`: returns the dimensions of the work grid;
- `get_global_size`: returns the number of work-items in the grid;
- `get_num_groups`: returns the number of work-groups in the grid;
- `get_group_id`: returns the ID of the work-group;
- `get_local_size`: returns the size of the work-group;
- `get_local_id`: returns the local ID of the work-item in the current work-group;
- `get_global_id`: return the global ID of the work-item in the work-grid.

Each work-item is executed by a compute element; each work-group is executed by a compute unit. Several work-groups can reside in a compute unit.

The memory model of OpenCL also resembles the CUDA one: beside the host memory, the device include shared, private, local, global and constant memories.

CPU-GPU transmissions make use of objects. A program object contains the binary source code of the kernel; a memory object is a buffer allocated to exchange data.

In the kernel definition, address spaces use qualifiers to define their memory type.

```
__constant float v1[4];

__kernel void kernelFunction(__constant int *v2,
                             __local int *v3,
                             __global int* v4) {
    int i; // __private
    __global int *v5;
    /*...*/
}
```

Work-items within the same work-group can be synchronized using `barrier`.

A platform is a specific OpenCL implementation of the APIs. `clGetPlatformIDs` and `clGetPlatformInfo` are used to access the available platforms and their information.

Each platform contains one or more devices that can be queried with `clGetDeviceIDs` and `clGetDeviceInfo`.

Multiple contexts may be instantiated in a program. Each context is associated with one or more devices. A context is a container for devices, program and memory objects, kernels and command queues. A context is created with `clCreateContext` or `clCreateContextFromType`.

A command queue on a specific device can be created with `clCreateCommandQueue`.

The source code for a kernel is a text string, and it has to be compiled and loaded from its context. The kernel is an instance of the execution of a program object.

```
const char *prog_src =
    "__kernel void kernelFunction(/*...*/) {"
    "   /*...*/                             "
    "}                                      ";

// Build program object
cl_program prog = clCreateProgramWithSource(
    context, 1, prog_src, NULL, &err);

// Compiles the program for each device in the context
clBuildProgram(prog, 1, devices, "", NULL, NULL);

// Create kernel from program object
clCreateKernel(prog, "kernelFunction", &err);
```

To transfer data to the device, memory objects (buffers) have to be created and enqueued.

```
// Specify if read-only, write-only or read-write
cl_mem buffer = clCreateBuffer(
    context, CL_MEM_READ_ONLY, sizeof(float) * N, NULL, &err);

// Specify if synchronous or asynchronous
clEnqueueWriteBuffer(
    queue, buffer, CL_TRUE, 0, sizeof(float) * N, in_data, 0, NULL, NULL);
```

Before launching a kernel on a specified NDRange (grid) parameters have to be specified.

```
// Specify which parameter is targeted
clSetKernelArg(
    kernel, 0, sizeof(cl_mem), &buffer);

// Specify global and local work size
clEnqueueNDRangeKernel(
    queue, kernel, 1, NULL, N, 32, 0, NULL, NULL);
```

After kernel execution, data has to be transferred back from device to host, and all resources are to be freed.

```
clEnqueueReadBuffer(
    queue, buffer, CL_TRUE, 0, sizeof(float) * N, results, 0, NULL, NULL);

clReleaseObject(buffer);
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseContext(context);
```

Command synchronization in queues is achieved with `clEnqueueBarrier`, that blocks the execution of following commands in the queue until the ones before finish; another option can be the use `clFinish`, that blocks the host until the queue is empty.

A `cl_eventobject` (that obeys an event wait list) can be given to each `clEnqueue*` function as a parameter. `clWaitForEvent` can be used to let commands wait for specific events.

HSA

HSA foundation wanted to achieve an efficient execution on heterogeneous systems by allowing a CPU-GPU transparent usage.

- hUMA: heterogeneous unified memory architecture
- hQ: heterogeneous queuing
- HSAIL: HSA intermediate language

An HSA enabled SoC allows data sharing along all processors.

Transfers do not have to be explicit and values are moved autonomously on demand, allowing the use of pointers. The CPU can simply pass a pointer to the GPU, and after the computation the result can be directly read.

A heterogeneous queuing can be achieved by adding an AQL (architectural queuing layer).

Each computing unit can enqueue a task to another CU. A work stealing scheduler keeps the system balanced. Furthermore, the OS doesn't have to be involved in data transfers or task scheduling.

There is also no need for an explicit synchronization when launching new threads from the host.

HSA uses a low-level intermediate representation that is close to a machine ISA level.

Each vendor can achieve its independent compilation and distribution.