Getting started with CUDA
Part 3 - Kernel programming

Edwin Carlinet, Joseph Chazalon {`firstname.lastname@epita.fr`}

Fall 2023

EPITA Research Laboratory (LRE)

Slides generated on September 8, 2023

Kernel programming

(Reminder) 3 simple abstractions for a scalable programming model

CUDA is based at its core on 3 key abstractions:

- a hierarchy of thread groups
- shared memories
- barrier synchronization

This enables a CUDA program to be:

- partitionned in blocks
- run on devices with different computation resources

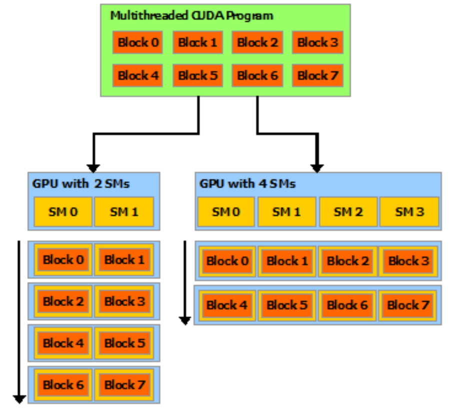

Figure 1: Automatic scaling

Several API levels

We now want to program kernels.
There are several APIs available:

- PTX assembly
- Driver API (C)
- Runtime C++ API ← **let us use this one**

We will first focus on the **language extensions** added to support kernel programming.
They are described in detail in Appendix B of the CUDA C Programming Guide.

Function Execution Space Specifiers

		Executed on the:	Only callable from the:
`__host__`	`float HostFunc()`	host	host
`__global__`	`void KernelFunc()`	device	host*
`__device__`	`float DeviceFunc()`	device	device

- `__global__` defines a kernel function
    - Each "`__`" consists of two underscore characters
    - A kernel function must return void
    - *It may be called from another kernel for devices of compute capability 3.2 or higher (Dynamic Parallelism support)
- `__device__` and `__host__` can be used together
- `__host__` is optional if used alone

Built-in Vector Types (1/2)

They make is easy to work with data like images.

Alignement must be respected in all operations.

Type	Align.
char1, uchar1	1
char2, uchar2	2
char3, uchar3	1
char4, uchar4	4
short1, ushort1	2
short2, ushort2	4
short3, ushort3	2
short4, ushort4	8

Type	Align.
int1, uint1	4
int2, uint2	8
int3, uint3	4
int4, uint4	16
long1, ulong1	4 if sizeof(long) is equal to sizeof(int) 8, otherwise
long2, ulong2	8 if sizeof(long) is equal to sizeof(int) 16, otherwise
long3, ulong3	4 if sizeof(long) is equal to sizeof(int) 8, otherwise
long4, ulong4	16

Type	Align.
longlong1, ulonglong1	8
longlong2, ulonglong2	16
longlong3, ulonglong3	8
longlong4, ulonglong4	16
float1	4
float2	8
float3	4
float4	16
double1	8
double2	16
double3	8
double4	16

Built-in Vector Types (2/2)

They all are structures.

They all come with a constructor function of the form `make_<type name>`:

```
int2 make_int2(int x, int y);
```

The 1st, 2nd, 3rd, and 4th components are accessible through the fields x, y, z, and w, respectively.

```
uint4 p = make_uint4(128, 128, 128, 255);
// or uint4 p(128, 128, 128, 255);
uint r = p.x, g = p.y, b = p.z, a = p.w;
```

`dim3` is an alias of `uint3` for which any component left unspecified is initialized to 1. Used to specify grid and block sizes.

```
dim3 blockSize(32, 32);
```

Built-in Variables

Some variables are **pre-defined in a kernel** and can be used directly.

Name	Type	Description
gridDim	dim3	dimensions of the grid
blockIdx	uint3	block index within the grid
blockDim	dim3	dimensions of the block
threadIdx	uint3	thread index within the block
warpSize	int	warp size in threads

Example:

```
__global__ void MatAdd(float* A, float* B, float* C, int rows, int cols)
{
    int j = threadIdx.x + blockIdx.x * blockDim.x;
    int i = threadIdx.y + blockIdx.y * blockDim.y;
    if (i < rows && j < cols)
        C[i][j] = A[i][j] + B[i][j];
}
```

Memory Hierarchy

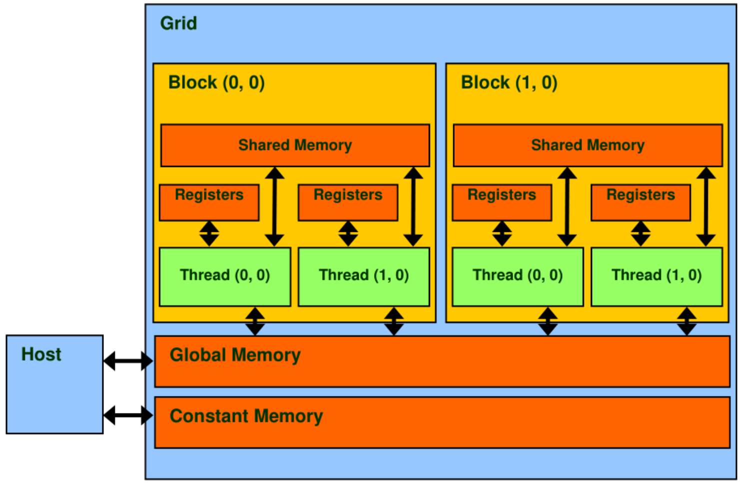

Figure 2: Programmer view of CUDA memories

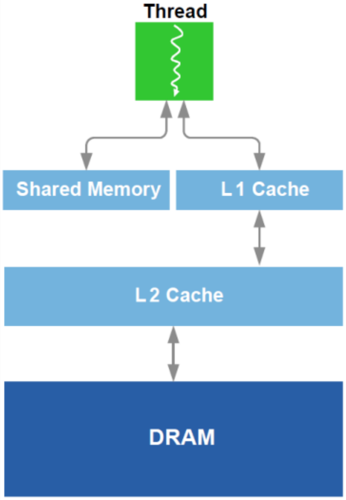

Figure 3: Cache hierarchy

Types of Memory

Registers	Used to store parameters, local variables, etc.
	Very fast
	Private to each thread
	Lots of threads $\Longrightarrow$ little memory per thread (spills in global memory if needed)
Shared	Used to store temporary data
	Very fast
	Shared among all threads in a block
Constant	A special cache for read-only values
	Slow at first then very fast
Global	Large and slow
	Shared among all threads in all blocks (in all kernels)
Caches	Transparent use
Local	*Local thread memory cached to L2 and/or L1*
	Ultimately stored in global memory if needed

Salient Features of Device Memory

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes[‡]	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	Yes[†]	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation

[†] Cached in L1 and L2 by default on devices of compute capability 6.0 and 7.x; cached only in L2 by default on devices of lower compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.

[‡] Cached in L1 and L2 by default except on devices of compute capability 5.x; devices of compute capability 5.x cache locals only in L2.

Cost to Access Memory

Space	Time	Notes
Register	0	
Shared	0	
Constant	0	Amortized cost is low, first access is high
Local	$>$ 100 clocks	
Parameter	0	
Global	$>$ 100 clocks	

Variable Memory Space Specifiers

How to declaring CUDA variables

Variable declaration	Memory	Scope	Lifetime
`int LocalVar;`	register	thread	thread
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

Remarks:

- `__device__` is optional when used with `__shared__`, or `__constant__`
- Automatic variables reside in a register

Where to declare variables?

Can host access it?

- Yes: **global** and **constant**
  Declare outside of any function
- No: **register** and **shared**
  Use or declare in the kernel

Example: Shared Variable Declaration

```
__global__ MatMulKernel(Matrix A, Matrix B, Matrix C, int rows, int cols)
{
  // ...
__shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
  // ...
}
```

Can also be declared to use dynamically allocated memory.
See the documentation for further details.

What can be shared, among what?

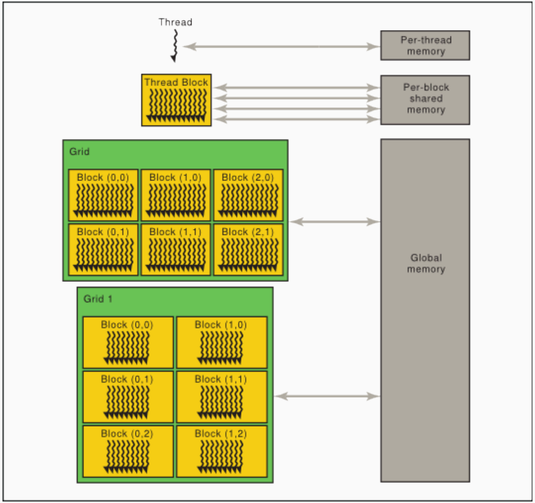

Figure 4: Memory sharing among threads, blocks and grids

Possible memory access:

- Among threads in the same grid (a kernel invocation):
  - Global memory

What can be shared, among what?

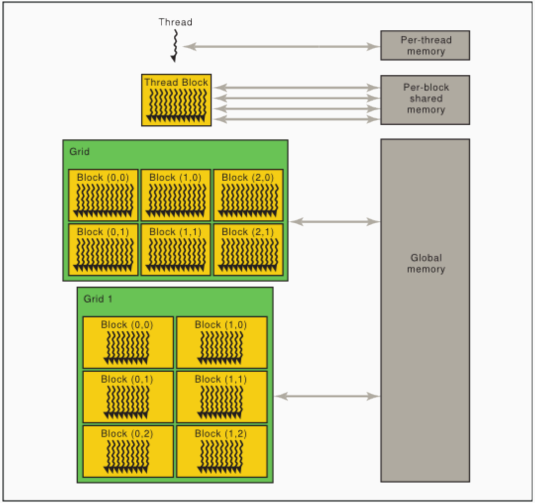

Figure 4: Memory sharing among threads, blocks and grids

Possible memory access:

- Among threads in the same grid (a kernel invocation):
  - Global memory
- Among threads in the same block:
  - Global memory
  - Shared memory (efficient)

What can be shared, among what?

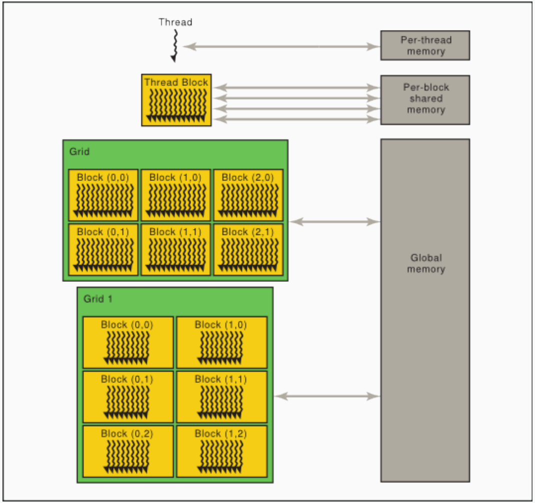

Figure 4: Memory sharing among threads, blocks and grids

Possible memory access:

- Among threads in the same grid (a kernel invocation):
  - Global memory
- Among threads in the same block:
  - Global memory
  - Shared memory (efficient)
- Per threads:
  - Global (not efficient)
  - Shared memory
  - Registers and local

Relaxed consistency memory model

The CUDA programming model assumes a device with a **weakly-ordered memory model**, that is the order in which a CUDA thread writes data to shared memory or global memory, is not necessarily the order in which the data is observed being written by another CUDA or host thread. *Think register/cache consistency, buffer flush. . .*

Example:

```
__device__ volatile int X = 1, Y = 2;
__device__ void write_from_thread1()
{
    X = 10;
    Y = 20;
}
```

```
__device__ void read_from_thread2()
{
    int A = X;
    int B = Y;
}
```

Possible outcomes for thread 2

Relaxed consistency memory model

The CUDA programming model assumes a device with a **weakly-ordered memory model**, that is the order in which a CUDA thread writes data to shared memory or global memory, is not necessarily the order in which the data is observed being written by another CUDA or host thread. *Think register/cache consistency, buffer flush...*

Example:

```
__device__ volatile int X = 1, Y = 2;
__device__ void write_from_thread1()
{
    X = 10;
    Y = 20;
}
```

```
__device__ void read_from_thread2()
{
    int A = X;
    int B = Y;
}
```

Possible outcomes for thread 2

Strongly-ordered memory model:

- $A = 1$ and $B = 2$
- $A = 10$ and $B = 2$
- $A = 10$ and $B = 20$

Relaxed consistency memory model

The CUDA programming model assumes a device with a **weakly-ordered memory model**, that is the order in which a CUDA thread writes data to shared memory or global memory, is not necessarily the order in which the data is observed being written by another CUDA or host thread. *Think register/cache consistency, buffer flush...*

Example:

```
__device__ volatile int X = 1, Y = 2;
__device__ void write_from_thread1()
{
    X = 10;
    Y = 20;
}
```

```
__device__ void read_from_thread2()
{
    int A = X;
    int B = Y;
}
```

Possible outcomes for thread 2

Strongly-ordered memory model:

- $A = 1$ and $B = 2$
- $A = 10$ and $B = 2$
- $A = 10$ and $B = 20$

Weakly-ordered memory model (like CUDA):

- All the previous
- And also $A = 1$ and $B = 20$!

Memory Fence Functions

Memory fence functions can be used to enforce some ordering on memory accesses.

```
void __threadfence_block();  // Among threads in a block
```

ensures that:

- All writes to all memory made by the calling thread before the call to `__threadfence_block()` are observed by all threads in the block of the calling thread as occurring before all writes to all memory made by the calling thread after the call to `__threadfence_block()`;
- All reads from all memory made by the calling thread before the call to `__threadfence_block()` are ordered before all reads from all memory made by the calling thread after the call to `__threadfence_block()`.

Like a flush of read and write queues.

```
void __threadfence();  // Among all threads in a grid
```

acts as `__threadfence_block()` but also ensure that threads from others blocks observe writes in order. **This requires to read an uncached value** and implies the use of the `volatile` keywords.

Synchronization Functions

```
void __syncthreads();
```

waits until all threads in the thread block have reached this point
and all global and shared memory accesses made by these threads prior to `__syncthreads()` are visible to all threads in the block.

Stronger than `__threadfence()` because it also synchronizes the execution.

`__syncthreads()` is used to coordinate communication between the threads of the same block.

`__syncthreads()` is allowed in conditional code but only if the conditional evaluates identically across the entire thread block, otherwise the code execution is likely to hang or produce unintended side effects.

Atomic Functions (1/2)

Atomic functions perform a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory.

Most of the atomic functions are available for all the numerical types:
`int`, `unsigned int`, `unsigned long long int`, `float`, `double`, `half`, etc.

Arithmetic functions

```
int atomicAdd(int* address, int val);
//int atomicSub(int* address, int val);
```

Read `old` at `address`, computes `(old + val)` and stores it back to `address`, returns `old`.

```
int atomicExch(int* address, int val);
```

Read `old` at `address`, stores `val` to `address`, and returns `old`.

```
int atomicMin(int* address, int val);
// int atomicMax(int* address, int val);
```

Compute and store min (max).

Atomic Functions (2/2)

Arithmetic functions (cont'd)

```
unsigned int atomicInc(unsigned int* address, unsigned int val);
//unsigned int atomicDec(unsigned int* address, unsigned int val);
```

Computes `(((old == 0) || (old > val)) ? val : (old-1)`

```
int atomicCAS(int* address, int compare, int val);
```

Computes `(old == compare ? val : old)`

Bitwise functions

```
int atomicAnd(int* address, int val);
int atomicOr(int* address, int val);
int atomicXor(int* address, int val);
```

Hardware, API, developper views

The API enables task scheduling on homogeneous hardware

API logical units map to hardware units, enabling work division, parallelization, and compatibility.

		Hardware view			
		ALU/core	SIMD unit	SM	Device
API view	thread	✓			
	warp		✓		
	block			✓	
	grid				✓

Developers chose how to map their problem to API units

Everything is possible by default, but some choices are better than others in practice.

		API view			
		thread	warp	block	grid
Data view	pixel	✓	?		
	line	?	✓	?	
	tile	?	?	✓	?
	image			?	✓
	...				
Computation view	unit comparison	✓	?		
	wave propagation	? ✓	✓	?	
	...				

Designing kernels in practice

1. Split the work (based on some standard algorithm, ideally)
2. Assign the work to compute abstraction (e.g. 3 pixels for each thread, 3 × 1024 pixels per block. . .)
3. Properly call the kernel depending on the expected block/grid sizes it expects

Debugging, Performance analysis and Profiling

printf

Possible since Fermi devices (Compute Capability 2.x and higher).

Limited amount of lines:

- circular buffer flushed at particular times
- but **not** at program exit: must include call to `cudaDeviceSynchronize()` before exiting

Example:

```
#include <stdio.h>
__global__ void helloCUDA(float f) {
    if (threadIdx.x == 0)
        printf("Hello thread %d, f=%f\n",
               threadIdx.x, f) ;
}

int main() {
    helloCUDA<<<1, 5>>>(1.2345f);
    cudaDeviceSynchronize();
    return 0;
}
```

OUTPUT:

```
Hello thread 0, f=1.2345
```

Global memory write

To dump then inspect a larger amount of intermediate data.
Analysis code should be removed for production.

Example:

```
__global__ void mykernel(float *input, float *output, float *intermediate) {
    // ...
    intermediate[threadIdx.x] = intermediate_result;
    // ...
}

int main() {
    // allocate input, output AND intermediate
    // ...
    mykernel<<<GS, BS>>>(input, output, intermediate);
    // ...
    // analyse intermediate results
    // ...
}
```

Check error messages

Did you check the error codes?

```
cudaError_t err = cudaMalloc((void **) &d_A, size);
if (err != cudaSuccess)  {
    printf("%s in %s at line %d\n",
           cudaGetErrorString(err),
           __FILE__,
           __LINE__);
    exit(EXIT_FAILURE);
}
```

CUDA tools

CUDA-GDB debugger

Debugging flags:

- `-g`: include host debugging information
- `-G`: include device debugging information
- `-lineinfo`: include line information with symbols

Based on GDB.

CUDA-MEMCHECK memory debugging tool

- No recompilation necessary
  ```
  cuda-memcheck myprogram
  ```
- Can detect the following errors: memory leaks, memory errors (like alignment issues), race conditions, illegal barriers...

nvprof profiler

```
nvprof myprogram
```

NSight

- Visual tool
- Great visualization of profiling results
- Other tools integrated

Other tools

- *cuobjdump*: host and device obj disassemble and overview
- *nvdisasm*: advanced analysis of device binaries
- *nvprune*: prunes host object files and libraries to only contain device code for the specified targets