GPU Computing

E. Carlinet, J. Chazalon (firstname.lastname@lrde.epita.fr)

Oct 21

EPITA Research & Development Laboratory (LRDE)

Slides generated on October 8, 2021

From programming model to hardware parallelism

GPU memory model

From programming model to hardware parallelism

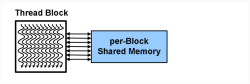
Programming Model (CUDA terminology)

Thread

- Instance of one *kernel* (list of instructions)
- If *active*, it has a *Program Counter*, *registers*, *private memory*, IO
- Thread ID = ID a *block*

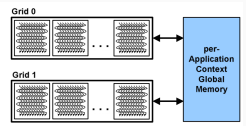
Block
A set of *threads* that cooperate:

- Synchronisation
- Shared memory
- Block ID = ID in a **grid**



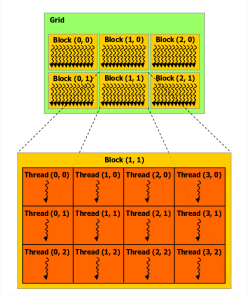
Grid
Array of blocks executing same *kernel*:

- Access to global GPU memory
- Sync. by stop and start a new kernel

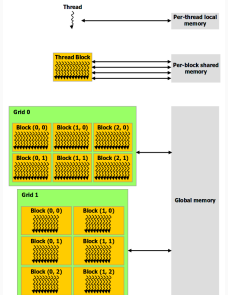


Programming Model - Summary

Hierarchy

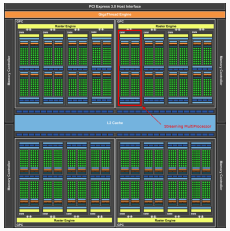


Memory



Mapping Programming model to hardware - the SMs

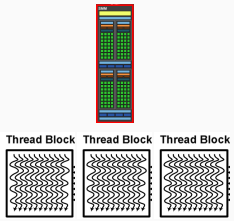
- CUDA's Thread/block/grid is mapped to GPU (N°Threads ≠ N°Cores)



- GTX 980 - 16 Streaming Multi-processors (SM)
- SMs are rather independant

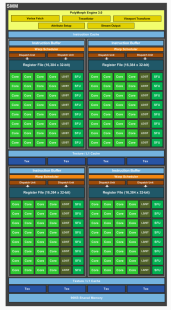
Mapping Programming model to hardware - the SMs

- *Any block* of *any grid* can be mapped to *any SM*



- Once *thread block* is affected to a SM, it won't move to another one.

Zoom on the SM



- SM organizes **blocks** into **warps**
- 1 warp = group of 32 threads

GTX 920:

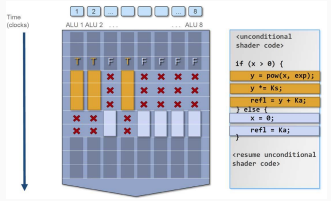
- 128 cores = 4 x 32 cores
- Quad warp scheduler selects **4** warps (**TLP**)
- And **2** independent instructions per warp can be dispatched each cycle (**ILP**)

Example:

- 1 (logical) *block* of 96 threads maps to: 3 (physical) *warps* of 32 threads

The SIMT Execution Model on CUDA Cores

- Divergent code paths (branching) pile up!



A mask allow to dis/activate threads:

If branch	1 1 0 1 0 0 0 0
Else branch	0 0 1 0 1 1 1 1

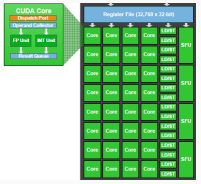
The SIMT Execution Model on CUDA Cores

What is the latency (in term of inst) of this code in the better and worst case ?

```
if a > 0:
    inst-a
    if b > 0:
        inst-b;
    else
        inst-c
else:
    inst-d
```


Zoom on the CUDA cores

1 core = 1 thread



- A *warp* executes 32 threads on the 32 CUDA cores
- The threads executes **the same** instruction (**DLP**)
- All instructions are SIMD (width = 32) instructions

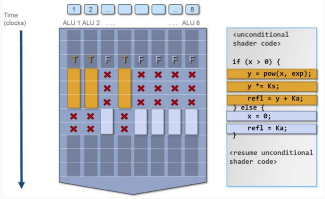
Each core:

- Floating point & Integer unit
- Fused multiply-add (FMA) instruction
- Logic unit
- Move, compare unit
- Branch unit
- The first IF/ID of the pipeline is done by the SM

Warning: SIMT allows to specify the execution and branching behavior of a single thread but maps to SIMD processors!

The SIMT Execution Model on CUDA Cores

- Divergent code paths (branching) pile up!



The SIMT Execution Model on CUDA Cores

What is the latency (in term of inst) of this code in the better and worst case ?

```
if a > 0:
    inst-a
    if b > 0:
        inst-b;
    else
        inst-c
else:
    inst-d
```

- Best case: `a > 0` is false for every thread. For all threads: `inst-d`
- Worst case: `a > 0` and `b > 0` is true for some but not all threads. For all threads:

```
inst-a
inst-b
inst-c
inst-d
```


The SIMT Execution Model on CUDA Cores

Loops

```
i = 0
while i < thread_id:
    i += 1
```

The SIMT Execution Model on CUDA Cores

Loops

```
i = 0
while i < thread_id:
    i += 1
```

Unrollable loops cost = max iterations, ie:

- Keep looping until all threads exit
- Mask out threads that have exited the loop

Execution trace	T0	T1	T2	T3
`i = 0`	0	0	0	0
`i < tid`	0	1	1	1
`i++`	0	1	1	1
`i < tid`	0	0	1	1
`i++`	0	1	2	2
`i < tid`	0	0	0	1
`i++`	0	1	2	3
`i < tid`	0	0	0	0

Occupancy: ILP vs TLP (1/2)

Occupancy

number of warps executed at the same time divided by the maximum number of warps that can be executed at the same time.

Generation	Warps per SM	Warps per scheduler	Issue rate	Issue width
G80	24	24	2	1
GT 200	32	32	2	1
Fermi	48	24	2	2
Kepler/Maxwell	64	16	1	2

Occupancy (warps/SM)	G80	GT200	Fermi	Kepler	Maxwell
100 %	24	32	48	64	64
50 % ...	12	16	24	32	32

Final note about terminology

	NVidia/CUDA	AMD/OpenCL	"CPU"
Thread	Cuda Processor	Processing Element	Lane
	Cuda Core	SIMD unit	Vector
GPU Core	Streaming Multiprocessor	Compute Unit	Core
	GPU Device	GPU Device	Device

GPU memory model

Occupancy: ILP vs TLP (1/2)

Occupancy

number of warps executed at the same time divided by the maximum number of warps that can be executed at the same time.

Generation	Warps per SM	Warps per scheduler	Issue rate	Issue width
G80	24	24	2	1
GT 200	32	32	2	1
Fermi	48	24	2	2
Kepler/Maxwell	64	16	1	2

Occupancy (warps/SM)	G80	GT200	Fermi	Kepler	Maxwell
100 %	24	32	48	64	64
50 % ...	12	16	24	32	32

CUDA Best Practice

It is a common and recommended practice to launch many more thread blocks than can be executed at the same time.

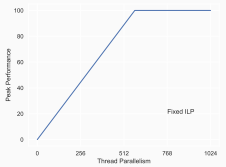
`No, increasing ILP is another way`

Occupancy: ILP vs TLP (2/2)

Check it on the GTX 480

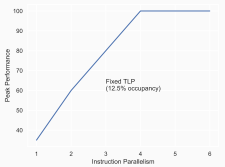
ILP = 1

```
#pragma unroll UNROLL
for( int i = 0; i < N_ITERATIONS; i++ )
{
    u = a * u + b;
}
```



ILP = 2 and more

```
#pragma unroll UNROLL
for( int i = 0; i < N_ITERATIONS; i++ )
{
    u = a * u + b;
    v = a * v + b;
    ...
}
```



Computation cost vs. memory cost

- Power measurements on NVIDIA GT200

	Energy/op (nJ)	Total power (W)
Instruction Control	1.8	18
Mult-add 32-wide warp	3.6	36
Load 128B from DRAM	80	90

With the same amount of energy:

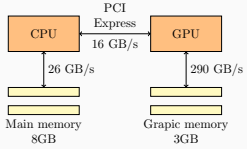
- Load 1 word from external memory (DRAM)
- Compute 44 flops

→ Must optimize memory first

External memory: discrete GPU

Classical CPU-GPU model

- Split memory space
- Highest bandwidth from GPU memory
- Transfers to main memory are slower

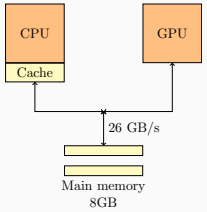


Intel i7 4770 / GTX 780

External memory: embedded GPU

Most GPUs today:

- Same memory
- May support memory coherence (GPU can read directly from CPU caches)
- More contention on external memory

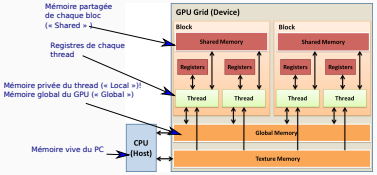


GPU: on-chip memory

Cache area in CPU vs GPU:

Figure 1-2. The GPU Devotes More Transistors to Data Processing

Memory model hierarchy (software)



Memory	On chip	Cached	Access	Scope	Lifetime
Register	✓	n/a	RW	1 thread	Thread
Local	✗	✓	RW	1 thread	Thread
Shared	✓	n/a	RW	All threads block	Block
Global	✗	✓	RW	All threads + host	Host
Constant	✗	✓	R	All threads + host	Host
Texture	✗	✓	R	All threads + host	Host

Non-aligned and strided load in L1 & L2

Q: how to make aligned-load with 2D-arrays ?

`ima(x,y) = y * width + x`

GPU: on-chip memory

Cache area in CPU vs GPU:

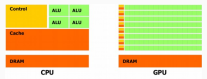
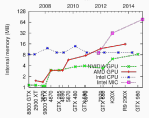


Figure 1-2. The GPU Devotes More Transistors to Data Processing

But if we include registers:

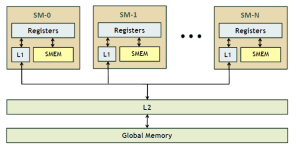
GPU	Registers files + caches
NVidia Maxwell	8.3 MB
AMD Hawaii GPU	15.8 MB
Core i7 CPU	9.3 MB

	CPU	GPU
Register / Core	256	65K



→ GPU has many more registers but made of simpler memory

Memory model hierarchy (hardware)



Cache hierarchy:

- Keep frequently-accessed data Core
- Reduce throughput demand on main memory L1
- Managed by hardware (L1, L2) or software (shared memory)

- On CPU, caches are designed to avoid memory latency
- On GPU, multi-threading deals with memory latency

Non-aligned and strided load in L1 & L2

Q: how to make aligned-load with 2D-arrays ?

`ima(x,y) = y * width + x`

To have coalesced access, you need to have a multiple of 32 (warp size) for:

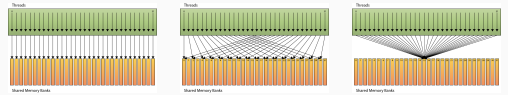
- the array width (it means *padding*)
- the thread block size

`ima(x,y) = y * stride + x`

Shared Memory & conflicts

Used a lot for local copy and fast-access

- Organized in memory bank (accessed concurrently)
- Every bank can provide 64 bits every cycle
- Only two modes:
  - Change after 32 bits
  - Change after 64 bits



No conflict | 2-way conflict | No conflict (broadcast)

Conflict = Serialized access (bad!)

Next time...

CUDA crash course