GPU Computing

Patterns for massively parallel programming (part 1)

The Global Memory

E. Carlinet, J. Chazalon {firstname.lastname@lrde.epita.fr}

April 22

EPITA Research & Development Laboratory (LRDE)

Slides generated on April 13, 2022

Course Agenda (2022-04)

1. *GPU and architectures* (2h, 6th AM)
2. *Programming GPUs with CUDA* (2h, 6th AM)
3. *TP 00 CUDA (Getting started)* (3h, 6th PM)
4. *Efficient programming with GPU (part 1)* (2h, 13th AM)
5. *TP 01 CUDA (Mandelbrot)* (3h, 13th PM)
6. *Efficient programming with GPU (part 2)* (2h, 20th AM)

   Project presentation (15 minutes, TBD)

Programming patterns & Memory Optimizations

Programming patterns & Memory Optimizations

The Programming Patterns

- Map
- Map + Local reduction
- Reduction
- Scan

The IP algorithms

- LUT Application
- Local features extraction
- Histogram
- Integral images

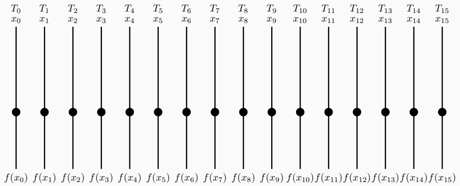
Map Pattern

Map Pattern Overview

Map replicates a function over every element of an index set

The computation of each pixel is independent w.r.t. the others.

```
out(x,y) = f( in(x,y) )
```

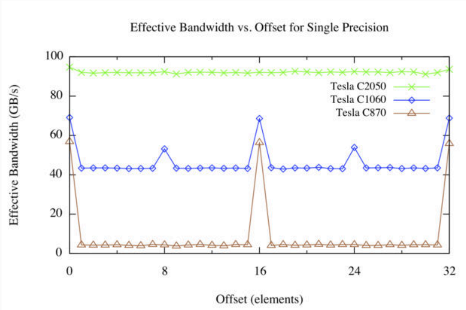


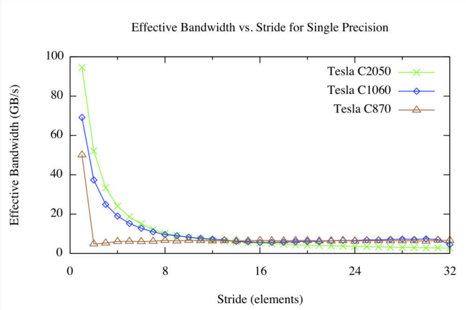
Nothing complicated but take care of memory access pattern.


```
void plus_one(int* a, int size, int k) {
  int i = blockDim.x * blockIdx.x +
          threadIdx.x + k;
  if (i < size)
    a[i] = a[i] + 1;
}
```

```
void plus_one(int* a, int size, int k) {
  int i = blockDim.x * blockIdx.x +
          threadIdx.x * k;
  if (i < size)
    a[i] = a[i] + 1;
}
```

What do you think about **k**'s impact of the performance?

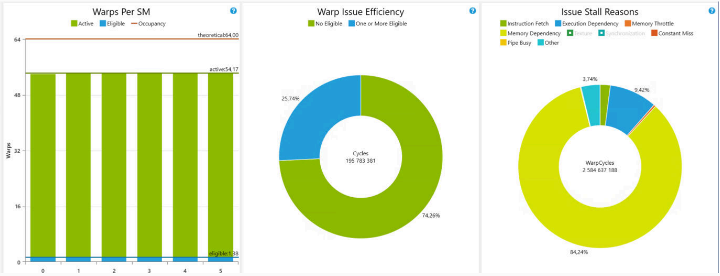




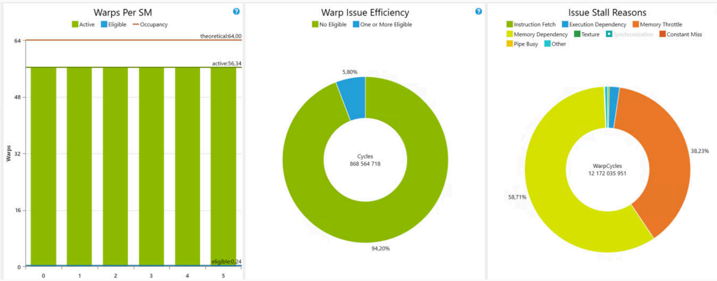
- Linear sequential access with offset (left) → 👍
- Strided access → 👎

Strided access pattern

- k = 1: 6.4 ms (with 600K values)



- k = 51: **28.7** ms



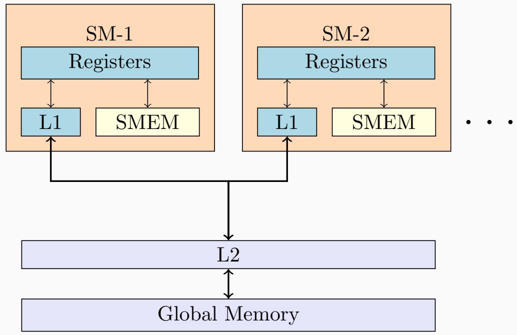
The Global Memory 🌶️🌶️

Memory Bandwidth

What you think about memory

Reality

Memory Access Hierarchy



GTX 1080 (Pascal)		Size	Bandwidth	Bus interface	Latency
L1 Cache (per SM)	Low latency	16 or 48K	1,600 GB/s	128 bits	10-20 cycles
L2 Cache		1-2M			
Global	High latency	8GB	320 GB/s	256~384 bits	400-800 cycles

Cached Loads from L1 low-latency memory (1/2)

Cached vs Un-cached

Two types of global memory loads: **Cached** (by default) or **Uncached** (L1 disabled)

Aligned vs Misaligned

A load is **aligned** if the first address of a memory access is multiple of 32 bytes

- Memory addresses must be type-aligned (ie `sizeof(T)`)
- Otherwise: poor perf (unaligned load)
- cudaMalloc = alignment on 256 bits (at least)

Coalesced versus uncoalesced

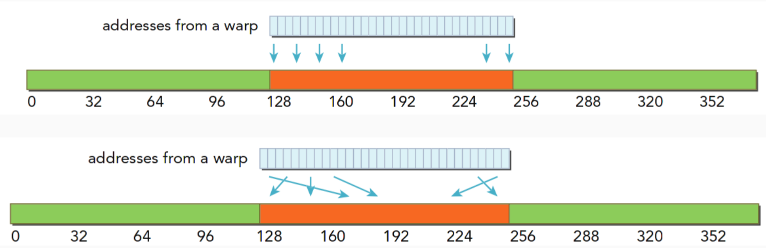
A load is **coalesced** if a warp accesses a contiguous chunk of data

- Minimize memory accesses caused by wrap threads
- Remind: all threads of a warp executes the same instruction
  → if a load, may be 32 different addresses

Cached Loads from L1 low-latency memory (2/2)

We need a load strategy:

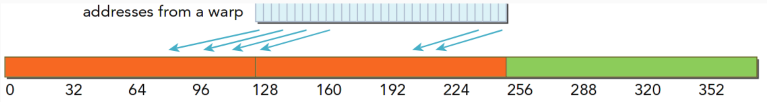
- 32 threads of warp access a 32-bit word = 128 bytes
- 128 bytes = L1 bus width (single load - bus utilization = 100%)
- Access permutation has no (or very low) overhead



Misaligned cached loads from L1

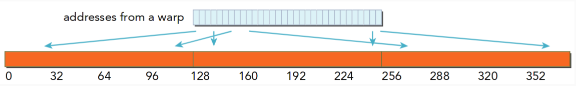
- If data are not 128-bits aligned, two loads are required

Adresses 96-224 required... but 0-256 loaded

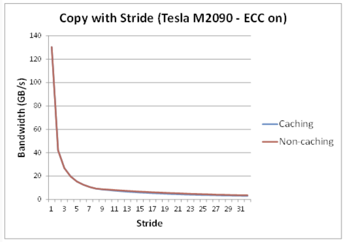


- If data is accessed strided

e.g. `u[2*k]`,...



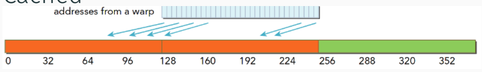
...cry !



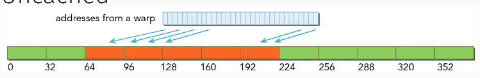
Loads from global (uncached) memory 🌶️🌶️

Same idea but memory is split in segments of 32 bytes

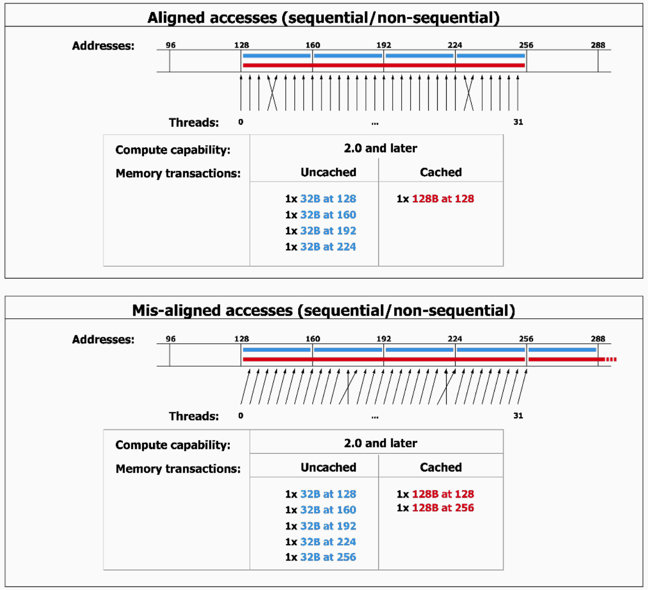
Cached



Uncached



Memory Access Summary 🌶️



Why would you need uncached memory ? 🌶️🌶️🌶️

Caching

- Better performance if non-coalesced access and data-reuse

Non-caching

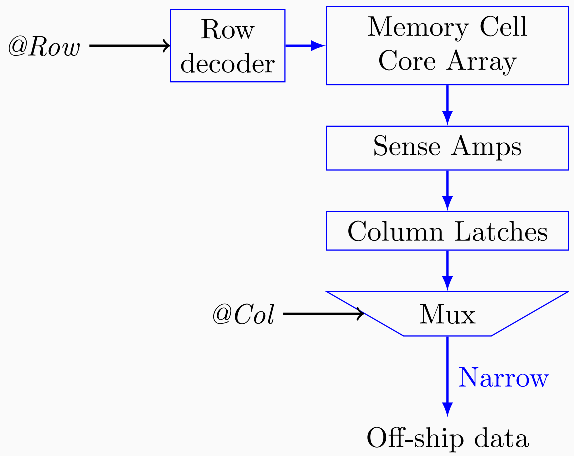
- Avoid wasting cache for one-time used data (stream usage) (→ more space for register spilling)

Memory architecture 🌶️🌶️🌶️

Memory architecture 🌶️🌶️🌶️

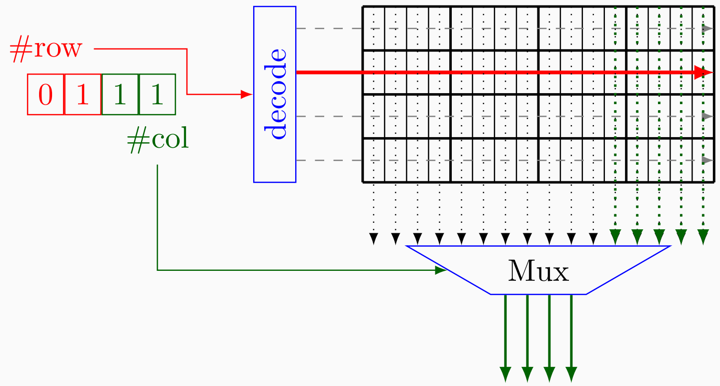
Why cacheline ? Why Alignement ?

- DRAM is organized in 2D Core arrays
- Each DRAM core array has about 16M bits



Example

- A 4 x 4 memory cell
- With 4 bits pin interface width



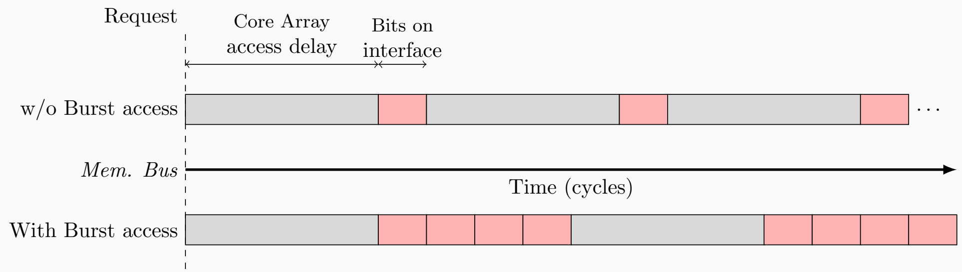
DRAM Burst

Reading from a cell in the core array is a very slow process (1/N th of the interface speed):

- DDR: Core speed = ½ interface speed
- DDR2/GDDR3: Core speed = ¼ interface speed
- DDR3/GDDR4: Core speed = ⅛ interface speed

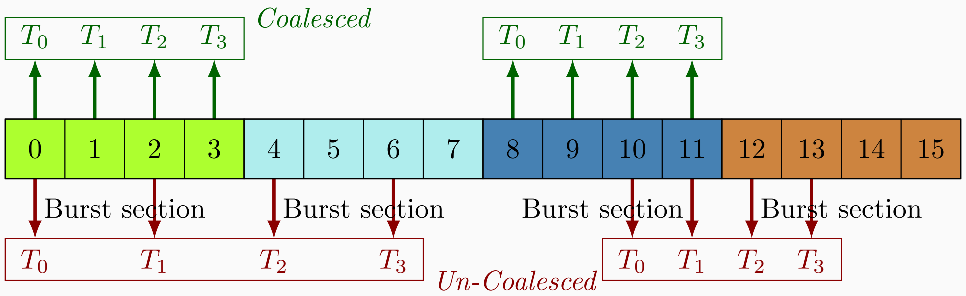
Solution: Bursting

Load N x interface width of **the same row**



Summary 🌶️

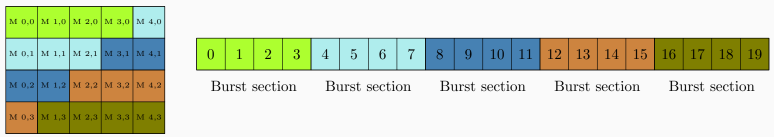
- Use **coalesced** (*contiguous* and *aligned*) access to memory:



- If all threads of a warp execute a load instruction into the same burst section → only one DRAM request 👍
- Otherwise:
  - Multiple DRAM requests are made
  - Some bytes transferred are not used

Better, but not enough to saturate the memory bus 👎 (will see later a solution).

Q: how to make coalesced/aligned loads with 2D-arrays ? 🌶️



1. Add padding to align rows on 256-bits boundaries
2. Thread reads data colomn-wise

```
ima(tid.x,tid.y) = tid.y * pitch + tid.x
```