GPU Computing

Patterns for massively parallel programming (part 1)

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

May 21

EPITA Research & Development Laboratory (LRDE)

Slides generated on 25 octobre 2021

Course Agenda (2021-10)

1. *GPU and architectures* (2h, Friday AM)
2. *Programming GPUs with CUDA* (2h, Friday PM)
3. *TP 00 CUDA (Getting started)* (3h, Monday or Tuesday)
4. *Efficient programming with GPU (part 1)* (2h, Wednesday AM)
5. *TP 01 CUDA (Mandelbrot)* (3h, Friday AM or PM)
6. *Efficient programming with GPU (part 2)* (2h, Monday 25th)
7. *Assignments* (remote, async, end of Oct. + Nov. 5th)
8. *TP01 Q/A* (3h, Friday, Nov. 5th)

Programming patterns & Memory Optimizations

Programming patterns & Memory Optimizations

The Programming Patterns

- Map
- Map + Local reduction
- Reduction
- Scan

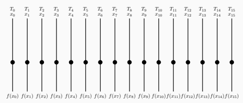
The IP algorithms

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

Map Pattern Overview

Map replicates a function over every element of an index set
The computation of each pixel is independant wrt the others.

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



Nothing complicated but take care of memory access pattern.

Map 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;
}
```

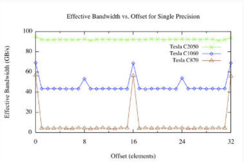
```
void plus_one(int* a, int size, int k) {
  int i = blockDim.x * blockIdx.x +
          threadIdx.x * k;
  if (i < size)
    a[i] = a[i] + 1;
}
```

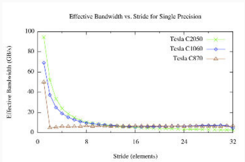
What do you think about k's impact of the performance ?


```
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 ?

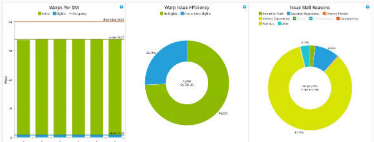




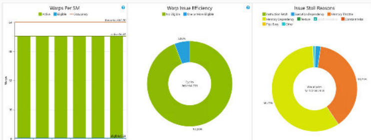
- Linear sequential access with offset (left) →
- Strided access →

Strided access pattern

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

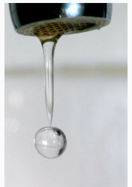


- k = 51 : 28.7 ms



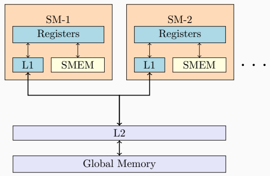
Memory Bandwidth

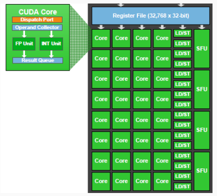
What you think about memory



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

Memory Performance Consideration

Memory Bandwidth

What you think about memory

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

Two types of global memory loads : **Cached** 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

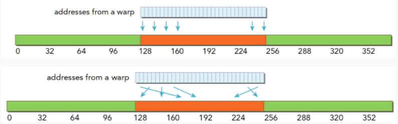
A load is **coalesced** if a warp accesses a contiguous chunk of data

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

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

We need a load stategy :

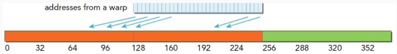
- 32 threads of warp access a 32-bit word = 128 bytes
- 128 bytes = L1 bus width (single load - bus utilization = 100%)
- Access permutation has no (or very low) overhead



Misaligned cached loads from L1

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

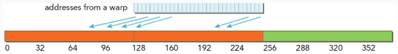
Adresses 96-224 required... but 0-256 loaded



Misaligned cached loads from L1

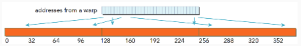
- If data are not 128-bits aligned, two loads are required

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



- If data is accessed strided

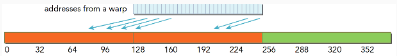
e.g. $u[2*k]$,...



Misaligned cached loads from L1

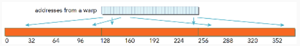
- If data are not 128-bits aligned, two loads are required

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

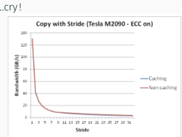


- If data is accessed strided

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



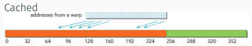
...cry!



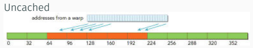
Loads from global (uncached) memory

Same idea but memory is split in segments of 32 bytes

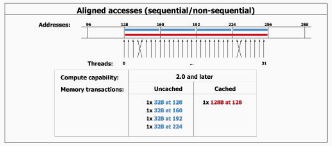
Cached

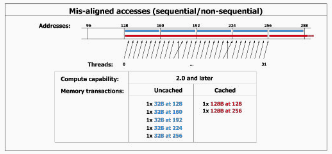


Uncached



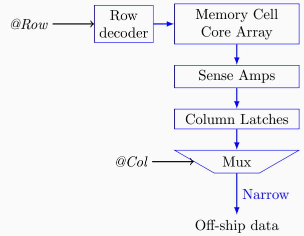
Coalesced Memory Access (Summary 1)





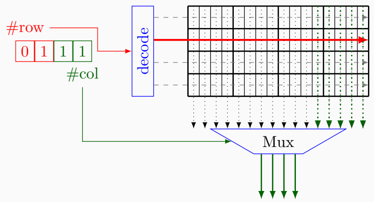
How memory works for real

- DRAM is organised in 2D Core array
- 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

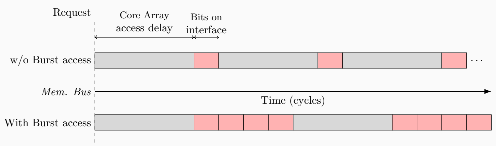
DRAM Burst

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

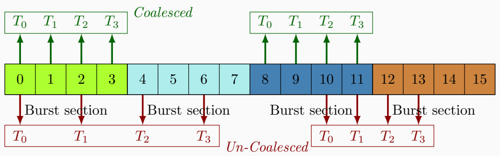
Load N x interface width of **the same row**



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

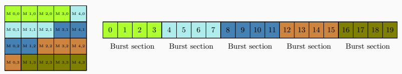
Summary

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

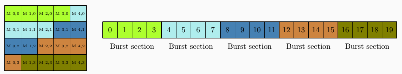


- If all threads of a warp execute a load instruction into the same burst section → only one DRAM request
- Otherwise :
  - Multiple DRAM requests are made
  - Some bytes transferred are not used

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

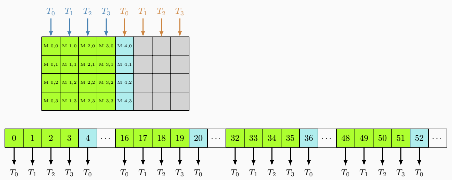


Q : how to make coalesced loads with 2D-arrays ?



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

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

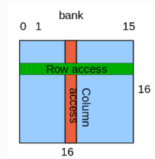


Using shared memory

Transposition

```
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdx.y * blockDim.y + threadIdx.y;

// transpose with boundary test
if (x < w && y < h)
  out[x * width + y] = in[y * width + x]
```

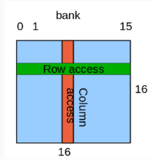


Where are non-coalesced access ?

Transposition

```
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdx.y * blockDim.y + threadIdx.y;

// transpose with boundary test
if (x < w && y < h)
  out[x * width + y] = in[y * width + x]
```



Where are non-coalesced access ?

→ $a[x][y]$

- Reads are coalesced
- Write are strided

Tiling and memory privatization in shared memory

For each block :

- read the tile from global to private block memory
- process the block
- write the tile from the private block memory to global memory

```
void mykernel() {
    __shared__ float private_mem[TILE_WIDTH][TILE_WIDTH];
}
```

Collaborative loading and writing when BLOCKDIM = TILEDIM

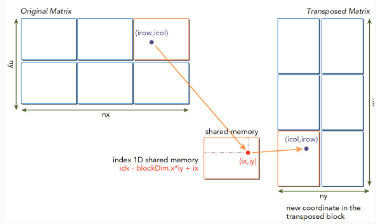
- All threads load one or more data
- Access must be **coelasced**
- Use barrier synchronization to make sure that all threads are ready to start the phase

```
void tiledKernel(unsigned char * in, unsigned char * out, int w, int h) {
  __shared__ float tile[TILE_WIDTH][TILE_WIDTH];

  int x = threadIdx.x + blockDim.x * blockIdx.x;
  int y = threadIdx.y + blockDim.y * blockIdx.y;
  // Load
  if (x < w && y < h)
    tile[threadIdx.y][threadIdx.x] = in[y * pitch + x];
  __syncthreads();
  // Process
  __syncthreads();
  // Write
  if (x < w && y < h)
    out[y * pitch + x] = tile[threadIdx.y][threadIdx.x];
}
```


Collaborative loading when BLOCKDIM < TILEDIM

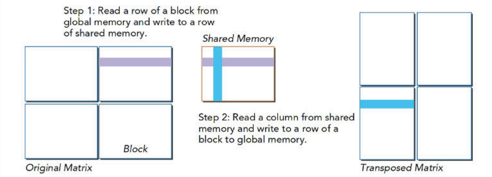
```
{
  __shared__ float tile[TILE_WIDTH][TILE_WIDTH];
  int* block_ptr = in + ...; // Compute pointer to the beginning of the tile
  for (int y = threadIdx.y; y < TILE_WIDTH; y += blockDim.y)
    for (int x = threadIdx.x; x < TILE_WIDTH; x += blockDim.x)
    {
      if (x < width && y < height)
        tile[y][x] = block_ptr[y * pitch + x];
    }
  __syncthreads();
}
```

Performance (GB/s on TESLA K40)

Copy (baseline)	Transpose Naive	Transpose Tiled
177.15 GB	68.98	116.82

Tiled transposition in shared memory (1/2)



Tiled transposition in shared memory (2/2)

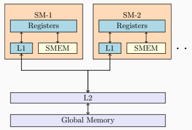
```
void tiledKernel(unsigned char * in, unsigned char * out, int w, int h) {
  __shared__ float tile[TILE_WIDTH][TILE_WIDTH];

  int x = threadIdx.x + blockDim.x * blockIdx.x; // src
  int y = threadIdx.y + blockDim.y * blockIdx.y; // src
  int X = threadIdx.x + blockDim.y * blockIdx.y; // dst
  int Y = threadIdx.y + blockDim.x * blockIdx.x; // dst

  // Load a line
  if (x < w && y < h)
    tile[threadIdx.y][threadIdx.x] = in[y * pitch + x];

  __syncthreads();
  // Write a line from a column in private mem
  if (x < w && y < h)
    out[Y * pitch + X] = tile[threadIdx.x][threadIdx.y];
}
```

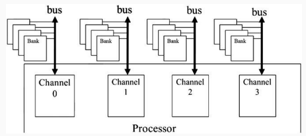

About shared memory

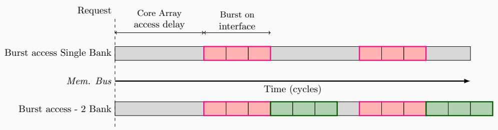


GTX 1080 (Pascal)		Size	Bandwidth	Latency
L1 Cache (per SM)	Low latency	16 or 48K	1,600 GB/s	10-20 cycles
L2 Cache		1-2M		
Global	High latency	8GB	320 GB/s	400-800 cycles

DRAM Banks

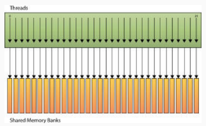
- Bursting : access multiple locations of a line in the DRAM core array (horizontal parallelism)
- 2 more forms of parallelism : channels & banks (vertical pipelining)
  1 processor has many channels (memory controller) with a bus that connects a set of DRAM banks (core array) to the processor.





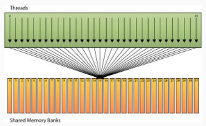
Bank conflicts in shared memory

- If 2 threads try to perform 2 **different** loads in the same bank → **Bank conflict**
- Every bank can provide 64 bits every cycle
- Only two modes :
  - Change after 32 bits
  - Change after 64 bits



`load DATA[tid.x]`

No Conflict

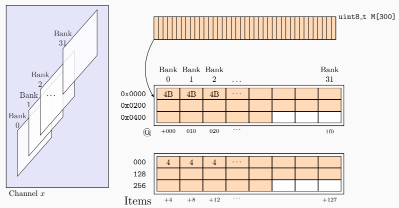


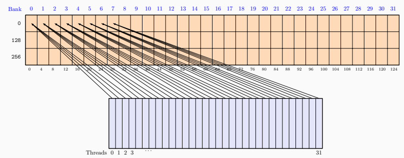
`load DATA[42]`

No conflict if loading the same address (broadcast)

Concrete Example for Shared Memory

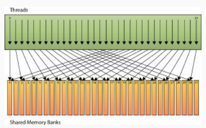
- Bank size : 4B = 4 uint8
- 32 Banks - Many Channels
- Warp Size = 32 threads



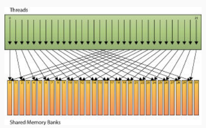


op	Items	Bank used	Conflict Free	#Loads
`load M[tid.x]`				

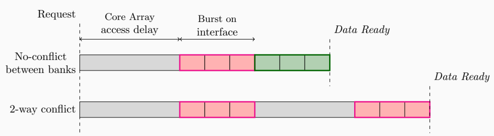
- 2-way conflicts

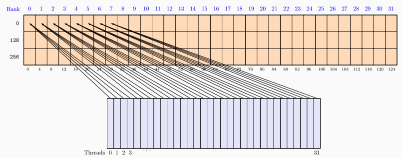


- 2-way conflicts

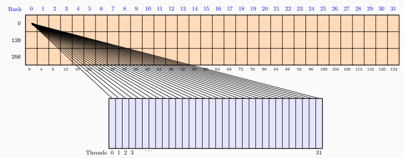


Conflict = Serialized access

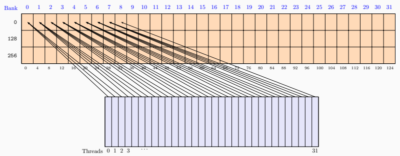




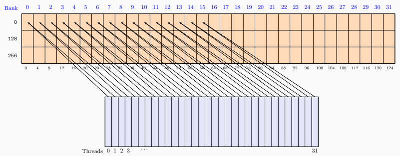
op	Items	Bank used	Conflict Free	#Loads
`load M[tid.x]`	`[0,1,...,31]`	`[0,1,...,7]`	✓	1x8
`load M[tid.x % 4]`				



op	Items	Bank used	Conflict Free	#Loads
`load M[tid.x]`	`[0,1,...,31]`	`[0,1,...,7]`	✓	1x8
`load M[tid.x % 4]`	`[0,1,2,3,0,1,2,3...]`	`[0]`	✓	1x1
`load M[tid.x + 1]`				

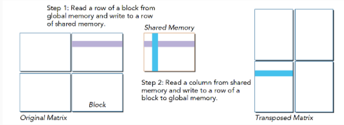


op	Items	Bank used	Conflict Free	#Loads
`load M[tid.x]`	`[0,1,...,31]`	`[0,1,...,7]`	✓	1x8
`load M[tid.x % 4]`	`[0,1,2,3,0,1,2,3...]`	`[0]`	✓	1x1
`load M[tid.x + 1]`	`[1,2,3,...32]`	`[0,1,...,8]`	✓	1x9
`load M[tid.x * 2]`				



op	Items	Bank used	Conflict Free	#Loads
`load M[tid.x]`	`[0,1,...,31]`	`[0,1,...,7]`	✓	1x8
`load M[tid.x % 4]`	`[0,1,2,3,0,1,2,3...]`	`[0]`	✓	1x1
`load M[tid.x + 1]`	`[1,2,3,...32]`	`[0,1,...,8]`	✓	1x9
`load M[tid.x * 2]`	`[0,2,4,...62]`	`[0,1,...,15]`	✓	1x16
`load M[tid.x * 8]`				

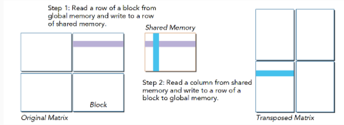
Bank conflicts in Transpose




```
__shared__ a[16][16];
Y = by*16+ty;
X = bx*16+tx;

a[y][x] = A[Y][X]
_syncthreads()
B[Y][X] = a[x][y];
```

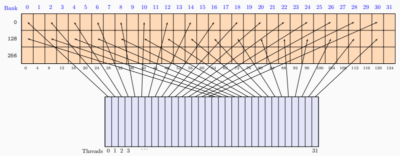

Bank conflicts in Transpose



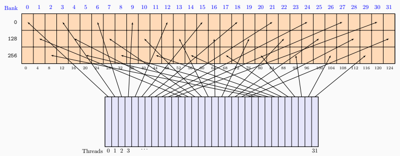

```
__shared__ a[16][16];
Y = by*16+ty;
X = bx*16+tx;

a[y][x] = A[Y][X]
_syncthreads()
B[Y][X] = a[x][y];
```

Reading a column may create bank conflicts



op	Items	Bank used	Conflict Free	#Loads
`load M[tid.x]`	`[0,1,...,31]`	`[0,1,...,7]`	✓	1x8
`load M[tid.x % 4]`	`[0,1,2,3,0,1,2,3...]`	`[0]`	✓	1x1
`load M[tid.x + 1]`	`[1,2,3,...32]`	`[0,1,...,8]`	✓	1x9
`load M[tid.x * 2]`	`[0,2,4,...62]`	`[0,1,...,15]`	✓	1x16
`load M[tid.x * 8]`	`[0,8,...,248]`	`[0,2,...,30]`	✗	2x16
`load M[tid.x * 12]`				



op	Items	Bank used	Conflict Free	#Loads
`load M[tid.x]`	`[0,1,...,31]`	`[0,1,...,7]`	✓	1x8
`load M[tid.x % 4]`	`[0,1,2,3,0,1,2,3...]`	`[0]`	✓	1x1
`load M[tid.x + 1]`	`[1,2,3,...32]`	`[0,1,...,8]`	✓	1x9
`load M[tid.x * 2]`	`[0,2,4,...62]`	`[0,1,...,15]`	✓	1x16
`load M[tid.x * 8]`	`[0,8,...,248]`	`[0,2,...,30]`	✗	2x16
`load M[tid.x * 12]`	`[0,12,...,372]`	`[0,1,...,31]`	✓	1x32

Solution to bank conflicts

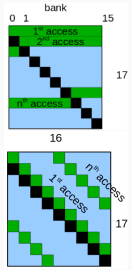
- With padding (to WRAP_SIZE + 1)

```
__shared__ a[17][17];
Y = by*16+ty;
X = bx*16+tx;

a[y][x] = A[Y][X]

_syncthreads()

B[Y][X] = a[x][y];
```



row & column access pattern

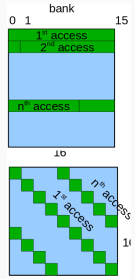
- Index mapping function
  `f: (x,y) → y * 16 + (x+y) % 16`

```
__shared__ a[...];
Y = by*16+ty;
X = bx*16+tx;

a[f(x,y)] = A[Y][X]

__syncthreads()

B[Y][X] = a[f(y,x)]
```



row & column access pattern

Performance (GB/s on TESLA K40)

Copy (baseline)	Transpose Naive	Transpose Tiled	Transpose Tiled+Pad
177.15 GB	68.98	116.82	121.83

Shared memory (Summary)

- Super fast access (almost as fast as registers)
- But limited ressources (64-96Kb by SM)

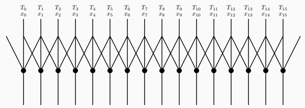
Use it carefully to avoid reducing the occupancy.

Stencil Pattern

The computation of a single pixel relies on its neighbors

Use case :

- Dilation/Erosion
- Box (Mean) / Convolution Filters
- Bilateral Filter
- Gaussian Filter
- Sobel Filter



Naive Stencil Implementation

Local average with a rectangle of radius r. (Ignoring border problems for now).

```
__global void boxfilter(const int* in, int* out, int w, int h, int r)
{
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  if (x < r || x >= w - r) return;
  if (y < r || y >= h - r) return;

  int sum = 0;
  for (int kx = -r; kx <= r; ++kx)
    for (int ky= -r; ky <= r; ++ky)
      sum += in[(y+ky) * w + (x+kx)];     // <==== \!/
  out[y * w + x] = sum / ((2*r+1) * (2*r+1));
}
```


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	Active threads limits
Maxwell (5.2)	64	16	2048
Pascal (6.1)	64	16	2048
Volta (7.0)	64	16	1024
Turing (7.5)	32	8	1024

Shared Memory ↗ → Number of ACTIVE Warp / SM ↘

Non-local Access Pattern with Tiling

Naive Stencil Performance

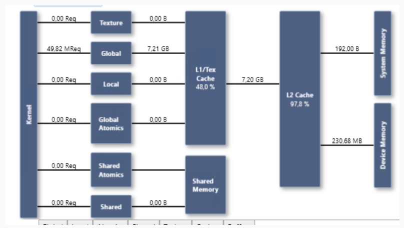
- Let's say, we have this GPU :
  Peak power : 1 500 GFlops and Memory Bandwith : 200 GB/s
- All threads access global memory
  - 1 Memory access for 1 FP Addition
  - Requires 1 500 × `sizeof(float)` = 6 TB/s of data
  - But only 200 GB/s mem bandwith → 50 GFLOPS (3% of the peak)

Compute-to-global-memory-access ratio

We need to have `#FLOP / #GlobalMemAccess >= 30` to reach the peak

Naive Stencil Performance

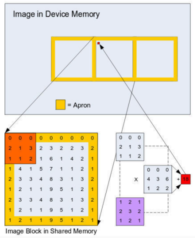
163 ms for a 24 MPix image



- Problem : too many access to global memory

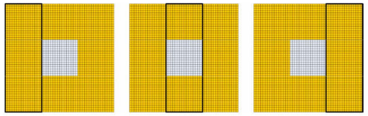
- Solution : tiling; copy data to shared memory per block first

Handling Border



1. Add border to the image to have in-memory access
2. Copy tile + border to shared memory

1. The bad way : ~~each thread copies one value and border threads are then idle~~.



2. The good way : a thread may copy several pixels

```
// TILE_WIDTH = blockDim.x + 2
__shared__ int tile[TILE_WIDTH][TILE_WIDTH]; // Alloc with the size of the block + b

int* block_ptr = in + ...;
for (int i = threadIdx.y; i < TILE_WIDTH; i += blockDim.y)
  for (int j = threadIdx.x; j < TILE_WIDTH; j += blockDim.x)
    data[i][j] = block_ptr[i * pitch + j];

__syncthreads();
```


Stencil Pattern with Tiling Performance

- Global memory : **163 ms** for a 24 MPix image
- Local memory : **116 ms** for a 24 MPix image (30% speed-up)