GPU Computing

Patterns for massively parallel programming (part 1)

Stencil Pattern and Shared Memory

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

April 22

EPITA Research & Development Laboratory (LRDE)

Slides generated on April 13, 2022

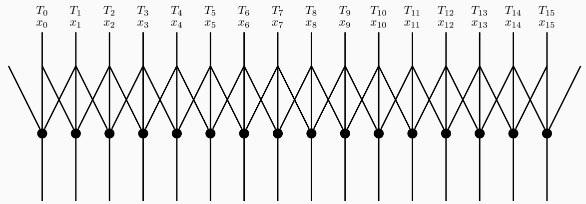
Non-local Access Pattern with Tiling 🌶️

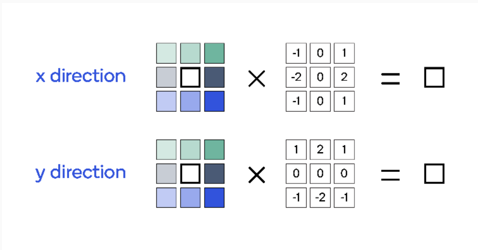
Stencil Pattern

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));
}
```


Naive Stencil Performance

- Let's say, we have this GPU:
  Peak power: 1 500 GFlops and Memory Bandwidth: 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) 👎

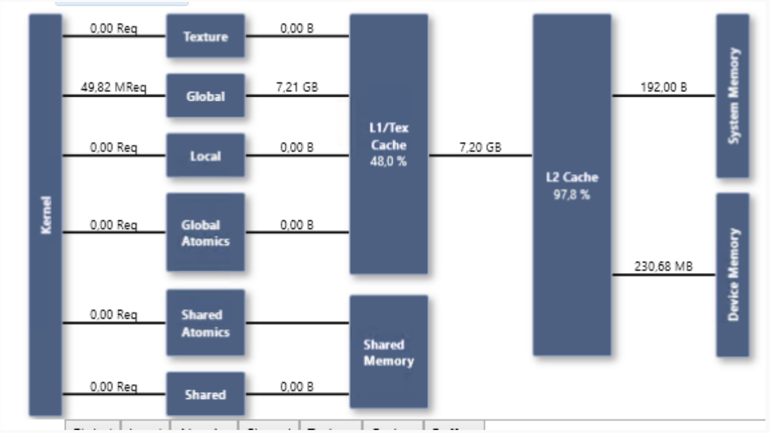
⚠️ Many programs are bandwidth-limited and not compute-limited

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

Tiling in shared memory 🌶️🌶️

Tiling and memory privatization in shared memory

Figure 1 - Tile subdivision used to render a frame

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 **coalesced**
- Use barrier synchronization to make sure that all threads are ready to start the phase

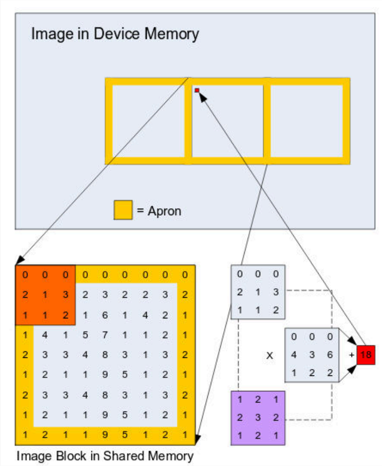
```
void tiledKernel(unsigned char * in, unsigned char * out, int w, int h) {
  __shared__ float tile[TILE_WIDTH][TILE_WIDTH];

  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)
```


Collaborative loading when BLOCKDIM < TILEDIM 🌶️🌶️🌶️

1 thread ↔ multiple loads

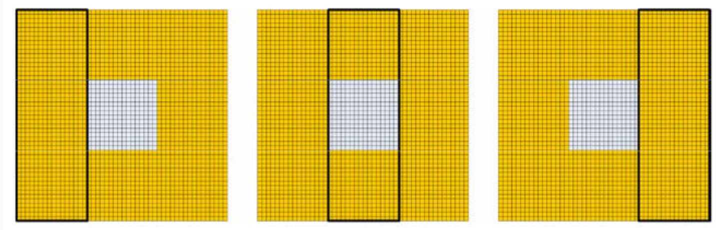
```
{
  __shared__ float tile[TILE_WIDTH][TILE_WIDTH];
  int* block_ptr = in + ...; // Compute pointer to the beginning of th
  for (int y = threadIdx.y; y < TILE_WIDTH; y += blockDim.y)
    for (int x = threadIdx.x; x < TILE_WIDTH; x += blockDim.x)
    {
      if (x < width && y < height)
        tile[y][x]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

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~~.

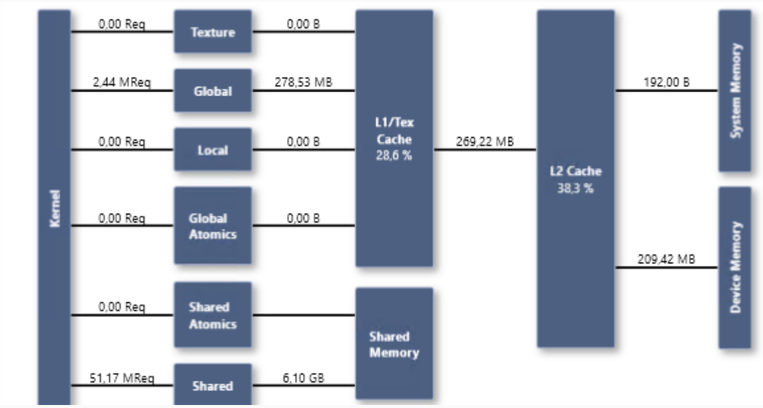


2. The good way: 1 thread ↔ multiple loads

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

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

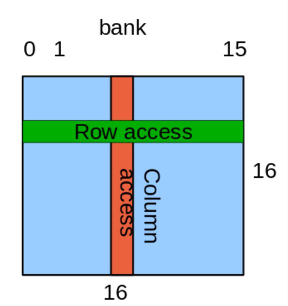
- Global memory: 163 ms for a 24 MPix image
- Local memory: 116 ms for a 24 MPix image (30% speed-up)



Using Shared Memory for Transposition

Naive 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]
```

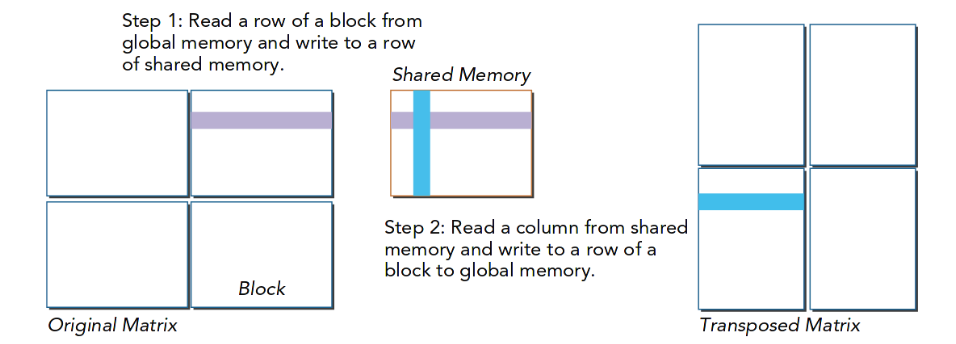


Where are non-coalesced access ?

→ a[x][y]

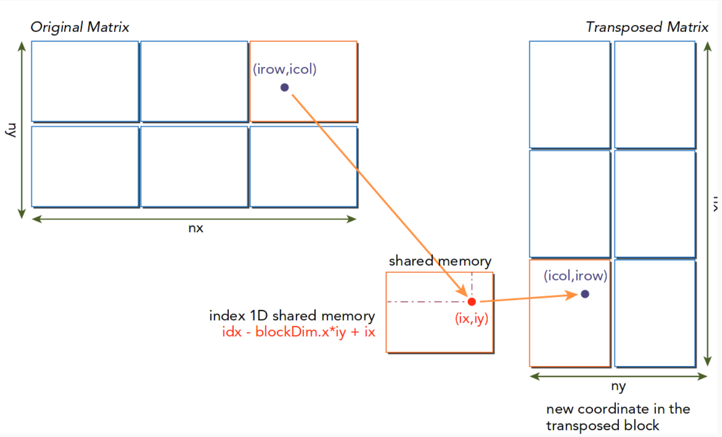
- Reads are coalesced
- Write are strided

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];
}
```

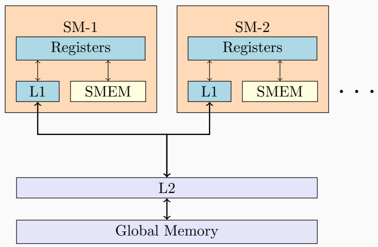
Performance (GB/s on TESLA K40)

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

Can we do better ?

Bank conflicts 🌶️🌶️🌶️

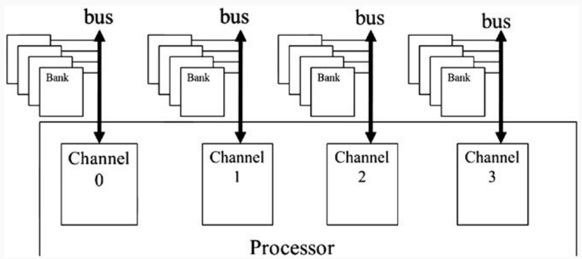
About shared memory

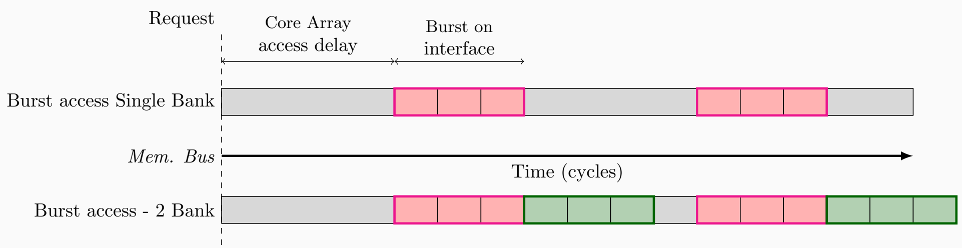


GTX 1080 (Pascal)		Size	Bandwidth	Latency
L1 Cache (per SM)	Low latency	16 or 48K	1,600 GB/s	10-20 cycles
L2 Cache		1-2M		
Global	High latency	8GB	320 GB/s	400-800 cycles

DRAM Banks

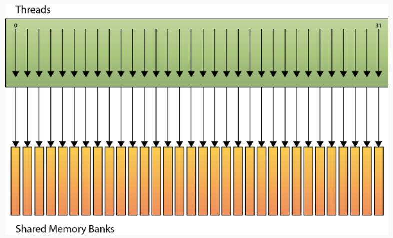
- Bursting: access multiple locations of a line in the DRAM core array (horizontal parallelism)
- 2 more forms of parallelism: channels & banks (vertical pipelining)
  1 processor has many channels (memory controller) with a bus that connects a set of DRAM banks (core array) to the processor.





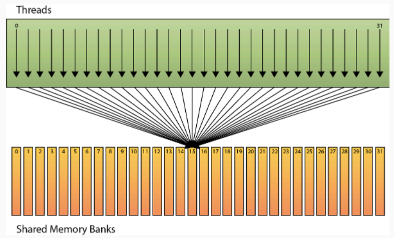
Bank conflits in shared memory

- 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]`

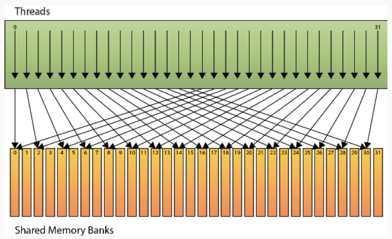
No Conflict



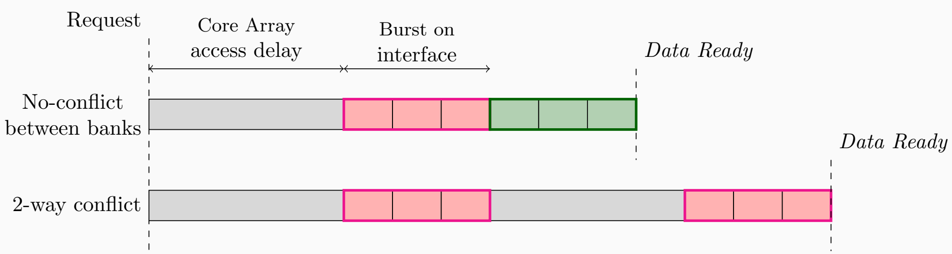
`load DATA[42]`

No conflict if loading the same address (broadcast)

- 2-way conflicts

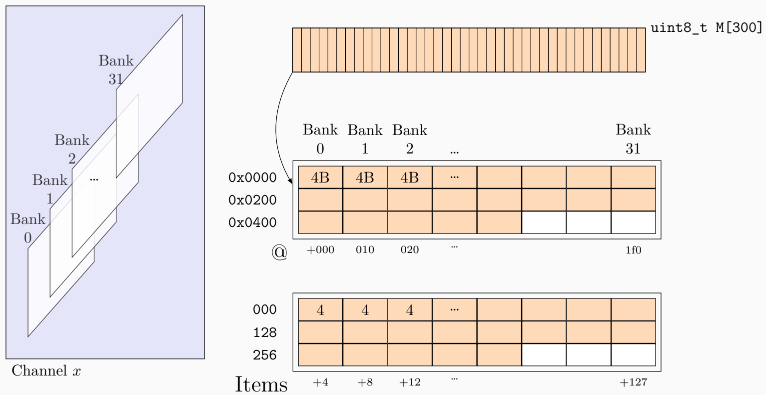


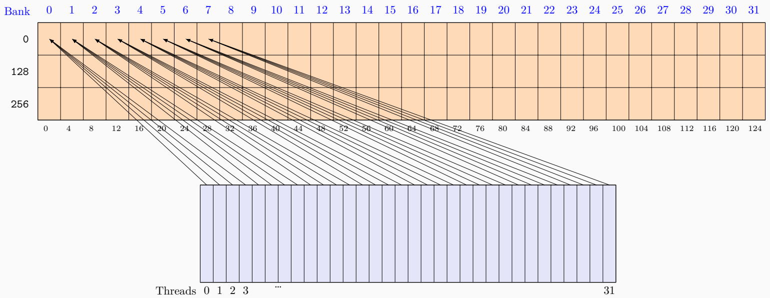
Conflict = Serialized access (👎)



Concrete Example for Shared Memory

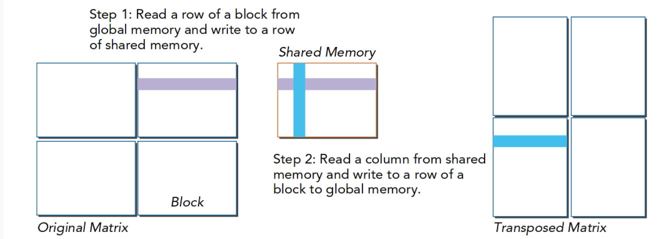
- Bank size: 4B = 4 uint8
- 32 Banks - Many Channels
- Warp Size = 32 threads





op	Items	Bank used	Conflict Free	#L
load M[tid.x]	[0,1,...,31]	[0,1,...,7]	✔	1x
load M[tid.x % 4]	[0,1,2,3,0,1,2,3...]	[0]	✔	1x
load M[tid.x + 1]	[1,2,3,...32]	[0,1,...,8]	✔	1x
load M[tid.x * 2]	[0,2,4,...62]	[0,1,...,15]	✔	1x
load M[tid.x * 8]	[0,8,...,248]	[0,2,...,30]	❌	2x
load M[tid.x * 12]	[0,12,...,372]	[0,1,...,31]	✔	1x

Bank conflicts in Transpose



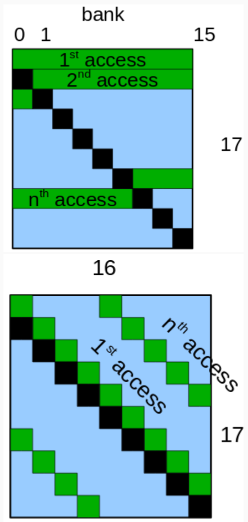

```
__shared__ a[16][16];
Y = by*16+ty;
X = bx*16+tx;

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

Reading a column may create bank conflicts

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];
```

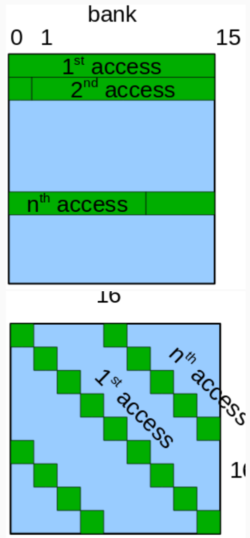


row & column access pattern

- 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

Conclusion 🌶️

Shared memory (Summary)

- Superfast access (almost as fast as registers) 👍
- Useful for block-wise collaborative computation (next course) 👍
- But limited resources (64~96Kb by SM) 👎

Use it carefully to avoid reducing the occupancy...

Occupancy

Number of active warps divided by the maximum number of warps that could be executed on the SM.

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

If Shared Memory usage ↗, then the number of ACTIVE Warp / SM ↘