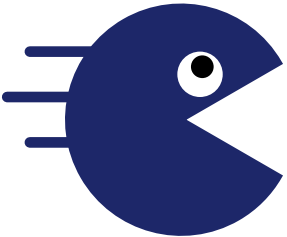


GPU Programming with CUDA

Polytech Sorbonne – EI5-SE – Calcul haute performance (EPU-F9-IHP)

Adrien Cassagne

January 12, 2026

Source of Inspiration

Acknowledgment

These document is strongly inspired by the excellent slides of the PAP class from the University of Bordeaux!

Please visit:

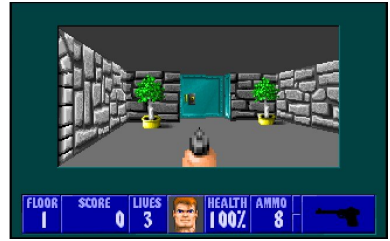
- https://gforgeron.gitlab.io/pap/
- https://raymond-namyst.emi.u-bordeaux.fr/ens/pap/PAP-GPU.pdf

Special thanks to Raymond Namyst and Pierre-André Wacrenier.

Table of Contents

Background – Part 1

- 1992: OpenGL 1.0 released by SGI
  - Multi-platform API for both 2D and 3D graphics
  - Initially aimed at Unix and quickly adopted for 3D gaming
- 1992: Wolfenstein 3D (Id Software)
  - First “First-Person Shooter”
- 1993: Birth of Nvidia
- 1995: Microsoft promotes its Direct3D API
  - But also supports OpenGL



Wolfenstein 3D

Background – Part 2

- 1995: 3Dfx interactive releases the Glide API
  - Subset of OpenGL 1.1
  - Geometry and texture mapping
- 1995: Nvidia NV1, first chip integrating
  - 3D rendering, video acceleration, GUI acceleration
  - But no native support of D3D triangular polygons (DirectX 1.0)
- 1995: ATI 3D Rage

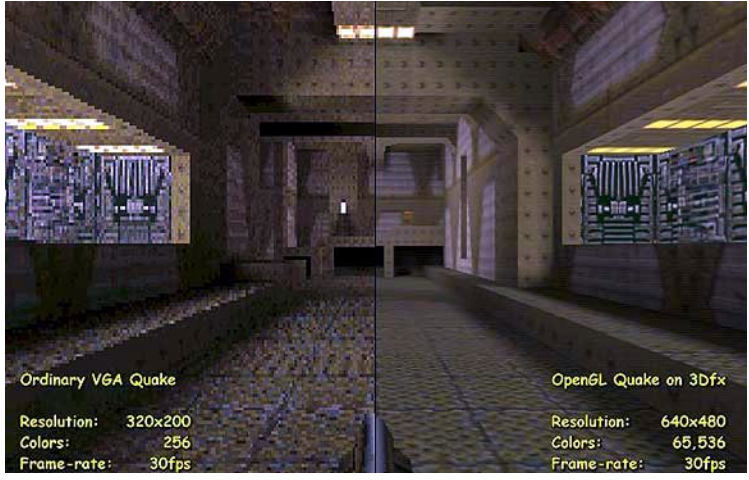
SEGA Virtua Fighter Remix for Diamond Edge3D (NV1)

Background – Part 3

- 1996: 3Dfx Voodoo Graphics
  - 3D only, Glide API
  - Killer app: Quake (ID software)
  - Beginning of a clear domination!
- 1997: ATI Rage Pro
  - AGP 2x interface (Intel)
  - 533 MB/s (against 132 MB/s using PCI)
  - NB: later, cards will embed fast GDDR memory
- 1997: Nvidia Riva 128 (Quake 2, Quake 3...)
- 1998: 3Dfx Voodoo 2
  - New landmark in framerates for many games
  - Scan Line Interleave (SLI) (aggregate multiple cards via a ribbon cable)

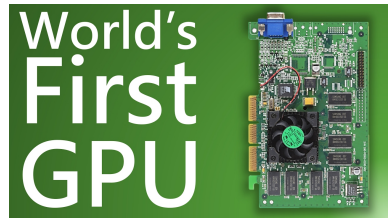
Tomb Raider (DOS, 1996)

Background – The Good Old Days...



Background – Part 4

- 1998: Sega choses PowerVR
  - Instead of 3Dfx for Dreamcast
- 1998: Microsoft Direct3D gains popularity
- 1998: 3Dfx decides to manufacture and sell their boards
  - Not competitive against ATI and Nvidia...
- 1999: Nvidia GeForce 256
  - First “Graphics Processing Unit”
  - Transformation and Lighting hardware engine



Nvidia GeForce 256

Background – Part 5

- 2001: Nvidia GeForce 3 (NV20)
  - Programmable units with mini-programs called “shaders”
  - Vertex (≈ 3D coordinates) and fragment (= pixel) shaders
- 2001: GPUs become General Purpose Accelerators (GPGPUs)
  - Texture can embed arbitrary data
  - Shaders can perform (almost) arbitrary computations
  - OpenGL can be used to perform scientific, numerical computations

Nvidia GeForce 3 (NV20)

Background – The Whole Story

Curious to find out more?

Please read the nice exhaustive history at TechSpot:

- https://www.techspot.com/article/650-history-of-the-gpu/

Boom of General Purpose GPU Computing

- May 2007: Nvidia foresees the potential market and releases the CUDA API
  - Compute Unified Device Architecture
  - Launches Tesla coprocessors
    - ECC memory
    - Double precision units
    - No video output!
- December 2007: AMD (formerly ATI)
  - Close To Metal API
  - Stream SDK
- 2007-today: Nvidia is the leader in GPU-accelerated computing

Nvidia Tesla (G80)

GPU Need Specific Programming Environments

- GPU feature many processors
  - 5000+ in Nvidia Tesla V100 (Volta)
  - At each cycle, many processors execute the same instruction on different data
    - "Simple Instruction – Multiple Data" execution model (SIMD)
    - GPUs require massive parallelism to achieve high performance
- GPU have on-board memory
  - Up to +32GB of GDDR
  - Data transfers between main memory and GPU embedded memory

Nvidia Tesla V100 (2017)

One Ring to Rule them All

- 2008: OpenCL
  - Khronos Compute Working Group (Apple, AMD, IBM, Qualcomm, Intel, Nvidia and many more)
  - Language + Library API
- OpenCL shares a lot of similarities with CUDA
  - But OpenCL is portable... even on non-GPU architectures
  - FPGA, Manycore processors

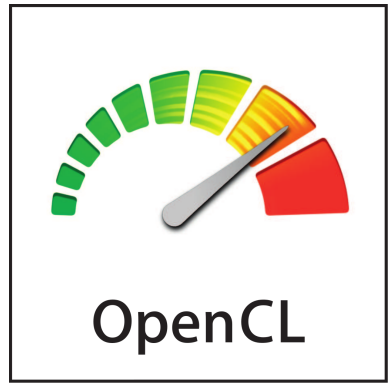
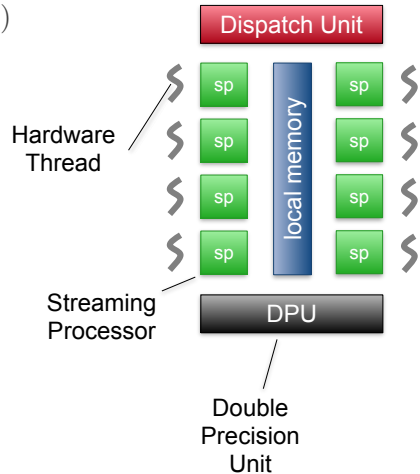


Table of Contents

2 Execution Model

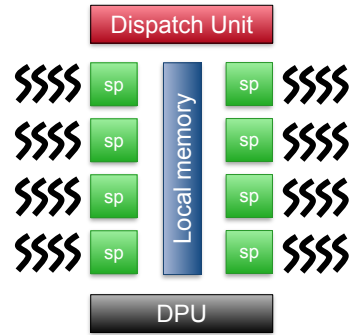
Introduction to the Nvidia Execution Model

- Basic block = Streaming Multiprocessor (SM)
  - Example on G80 SM (GeForce 8800 GTX)
- SM are clusters of 8 Streaming Processors
  - Local memory sharing
  - Synchronization
- Streaming Processor (SP)
  - 64 KB registers!
  - Threads are just “sets of registers”
    - Creation/destruction is free!
  - Interleaved execution of sequential hardware threads (up to 128 per SP)



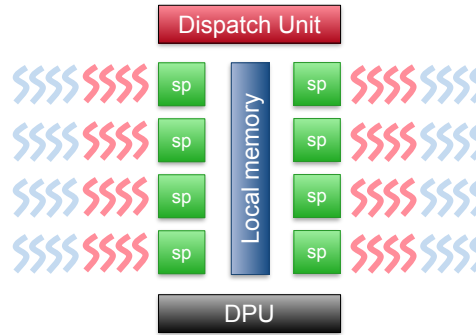
Single Instruction Multiple ~~Data~~ Threads

- Only one instruction dispatch unit per Streaming Multiprocessor
  - All SP execute the same instruction at the same clock cycle
  - On different data = Single Instruction Multiple Data (SIMD)
    - Nvidia call this SIMT (T for “Threads”)
- The Dispatch Unit takes 4 cycles to fetch & decode instructions
  - 4 sets of 8 threads are scheduled in a row, executing the same instruction
  - Context switch is free!



Warps and Half-warps

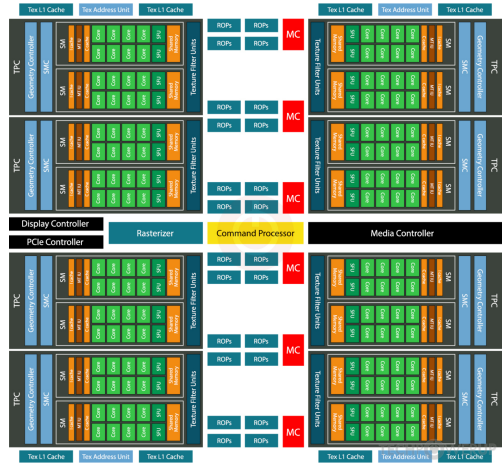
- Threads are implicitly grouped in “warps”
  - Warp = 32 threads (Nvidia)
  - On AMD GPUs, it is called a wavefront and it groups 64 threads
  - All threads of the same warp execute the same instruction at the same logical cycle
    - No divergence!
- Loading data from global memory is expensive
  - Therefore, more than 4 threads per SP are necessary
  - 128 threads are enough to hide memory latency



The Whole Picture

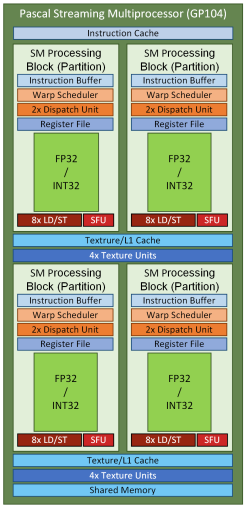
2 Execution Model

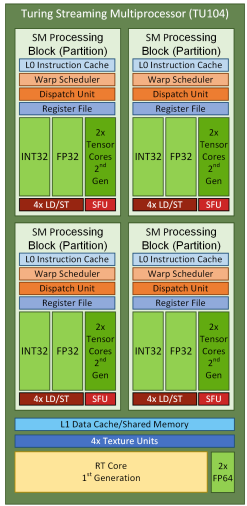
- GPU = set of Streaming Multiprocessors sharing a global memory
- Nvidia GeForce 8800 GTX ($\approx$ Tesla C870)
  - 16 SM
  - $8 \times 16 = 128$ CUDA cores
  - 128 threads max per processor = 16,384 threads!
- Not exactly the usual meaning of “thread”...
  - Data-parallelism
  - Regular access patterns

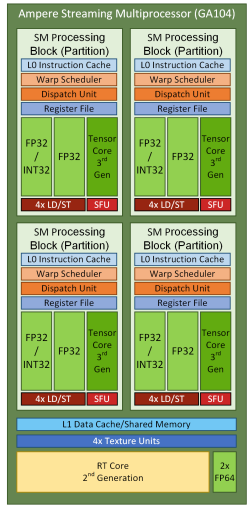


Modern Streaming Multiprocessors

2 Execution Model







- **Pascal** (2016, 14/16 nm)
  - 128 SP per SM
  - GTX 1080 Ti (20 SM)
  - **Jetson TX2** (2 SM)
- **Turing** (2018, 12 nm)
  - 64 SP per SM
  - RTX 2080 Ti (72 SM)
- **Ampere** (2020, 7/8 nm)
  - 128 SP per SM
  - RTX 3090 Ti (82 SM)
- **Ada Lovelace** (2022, 5 nm)
  - 128 SP per SM
  - RTX 4090 (128 SM)

Table of Contents

3 CUDA Programming Environment

Presentation

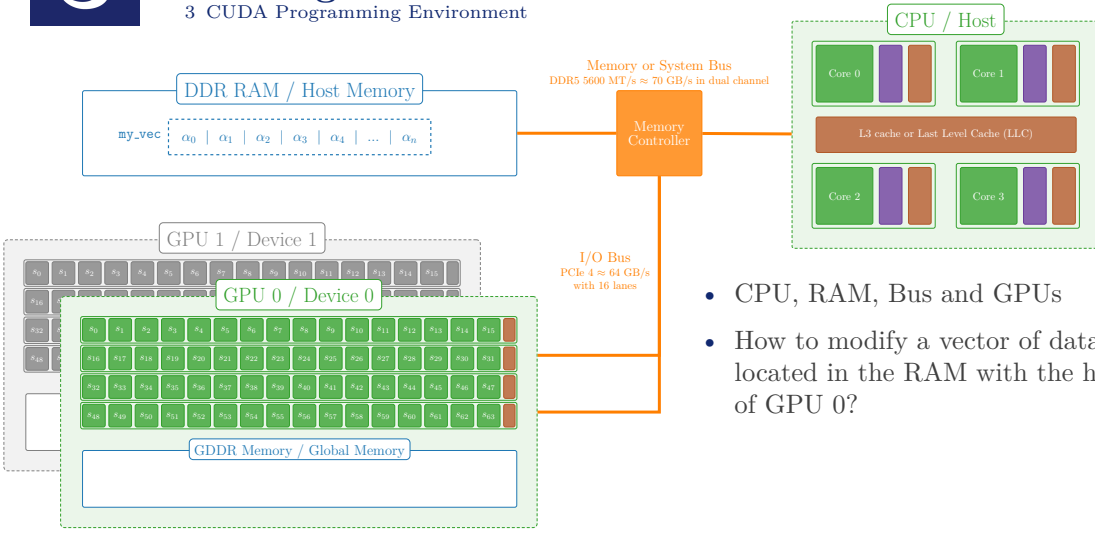
3 CUDA Programming Environment

CUDA is a language based on top of the C++, it requires a specific compiler to work: the *Nvidia CUDA Compiler* (or `nvcc`). `nvcc` targets either **host** code and **device** code.

- **Host code** runs on the **CPU**
  - `nvcc` relies on a traditional C++ compiler for this part (`g++`, `clang++`, ...)
  - Hardware discovery
  - Device (e.g. GPU) selection
  - On-device memory management
  - Memory transfers
  - GPU program launch
- **Device code** runs on the **GPU**
  - `nvcc` actually compiles the code that will run on the GPU
  - C++ language + a few keywords
  - Code is sent to device, and executed
  - Code entry points are named “kernels”
  - Kernel ≈ main function
    - Generally invoked from CPU side
  - Kernels are executed in parallel by many GPU threads

The Big Picture – Part 1

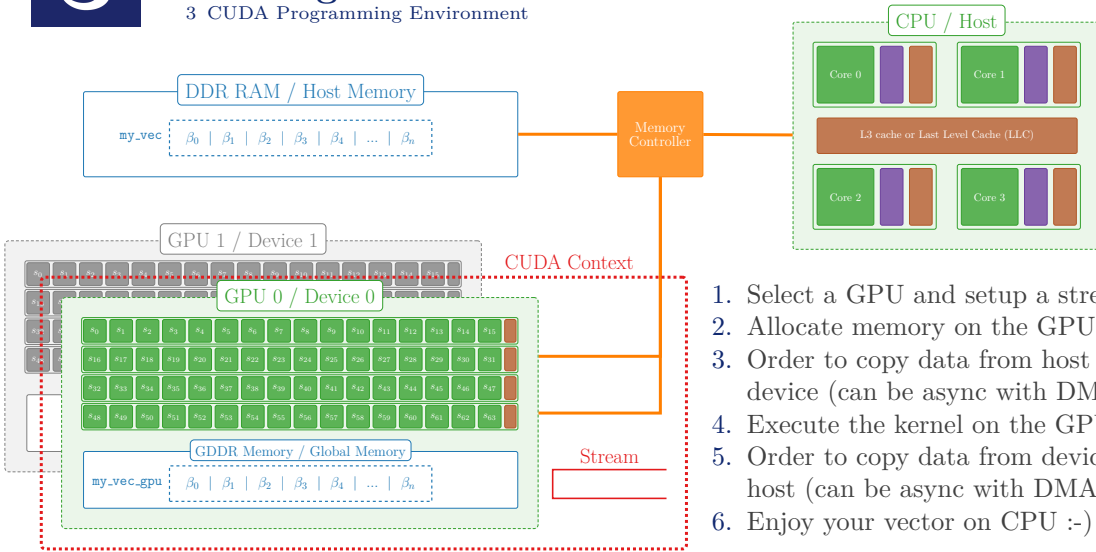
3 CUDA Programming Environment



- CPU, RAM, Bus and GPUs
- How to modify a vector of data located in the RAM with the help of GPU 0?

The Big Picture – Part 2

3 CUDA Programming Environment



1. Select a GPU and setup a stream
2. Allocate memory on the GPU
3. Order to copy data from host to device (can be async with DMA)
4. Execute the kernel on the GPU
5. Order to copy data from device to host (can be async with DMA)
6. Enjoy your vector on CPU :-)

CUDA Host – 1. Context and Stream

- Selecting **a context** is the same as picking **a GPU** in CUDA
- Selecting a GPU is not mandatory
  - If no GPU is selected, then the default one will be used
  - To force the selection, use: `cudaError_t cudaSetDevice(int device)`

```
1 #include <cuda.h>
2
3 // set device to be used for GPU executions (here the first GPU in the list)
4 cudaSetDevice(0);
```

- A **CUDA stream is a queue** where the host can push commands
- Commands are issued in the same order as they have been pushed (**FIFO**)
- A stream is implicitly dedicated to a context by default
  - This **default stream is synchronous** for each CUDA host operation
  - The runtime allows to create multiple streams to execute multiple kernels at the same time but it is not in the range of this introduction class

CUDA Host – 2. Memory Buffers Allocation

- CUDA provides its own primitive to allocate buffers on the GPU's global memory: `cudaError_t cudaMalloc(void** devPtr, size_t size)`

```
1  float *d_A, *d_B, *d_C;
2  size_t bytes = N * sizeof(float); // N is the number of float elements in arrays
3
4  // allocate memory for arrays d_A, d_B, and d_C on device
5  cudaMalloc(&d_A, bytes);
6  cudaMalloc(&d_B, bytes);
7  cudaMalloc(&d_C, bytes);
8
9  /* do something */
10
11 // free GPU memory
12 cudaFree(d_A);
13 cudaFree(d_B);
14 cudaFree(d_C);
```

CUDA Host – 3. Write from Host to Device


```
1  // allocate and initialize buffers on the host (CPU) memory space
2  float* h_A = malloc(sizeof(float) * N); // alloc
3  float* h_B = malloc(sizeof(float) * N); // alloc
4
5  // init host buffers
6  for (unsigned int i = 0; i < n_elements; i++) {
7    h_A[i] = i;
8    h_B[i] = (elements - 1) - i;
9  }
10
11 // copy data from host arrays h_A and h_B to device arrays d_A and d_B
12 cudaMemcpy(d_A, h_A, bytes, cudaMemcpyHostToDevice);
13 cudaMemcpy(d_B, h_B, bytes, cudaMemcpyHostToDevice);
```

- `cudaMemcpy` is always blocking, see `cudaMemcpyAsync` for non-blocking variant

CUDA Host – 4. Execute a GPU Kernel


```
1  // let's suppose we implemented the GPU kernel "add_vectors" with the interface below
2  __global__ void add_vectors(const float *A, const float *B, float *C);
```

```
1  // set execution configuration parameters
2  //      block_dim: number of CUDA threads per block
3  //      grid_dim: number of blocks in the grid
4  size_t block_dim = /* ??? */; // not very important now, we will see this later
5  size_t grid_dim = /* ??? */; // not very important now, we will see this later
6
7  // launch kernel
8  add_vectors<<< grid_dim, block_dim >>>(d_A, d_B, d_C);
```

- `kernel<<< x, y >>>(param1, param2, ...)` is the minimal CUDA syntax to invoke a kernel on GPU
  - This is not C++ compliant and requires the CUDA compiler to work
  - Invoking a kernel over the default stream is blocking (= synchronous)

CUDA Host – 5. Read from Device to Host


```
1  // allocate and initialize buffers on the host (CPU) memory space
2  float* h_C = malloc(sizeof(float) * N); // alloc
3
4  // copy data from device array d_C to host array h_C
5  cudaMemcpy(h_C, d_C, bytes, cudaMemcpyDeviceToHost);
6
7  // here you can use 'h_C' as usual on the host (CPU) side
8  for (unsigned int i = 0; i < N; i++) {
9    printf("h_C[%u] = %f\n", i, h_C[i]);
10 }
```

- Same `cudaMemcpy` primitive as for “host to device” copy is used
- This is, once again, a blocking call

CUDA Host API – Summary

- All the previous slides present C++ code executed by the CPU
  - The CPU (host) controls the GPU (device)
  - CUDA Host API is mainly a standard C++ library (except for the weird kernel invocation...)
- We did not see
  - Some mysterious arguments inside the `<<< ... >>>` of a kernel invocation
  - The code executed on GPU ➔ CUDA kernels

CUDA Device

- Contrary to the CUDA Host API, CUDA “device” programs target GPU
- It is an extension of the C++ language
  - New keywords: `__global__`, `__device__`, etc.
  - Special predefined structures like `blockIdx`, `blockDim`, `threadIdx`, ...
- CUDA device programs (= kernels) are stored in files with the `*.cu` extension
- A CUDA program must expose at least one “kernel” function
  - That is, a function that can be invoked from the CPU side
  - Can be seen as the traditional `main` function
  - But a CUDA program can expose multiple kernels

From Host to Device – Kernel Invocation

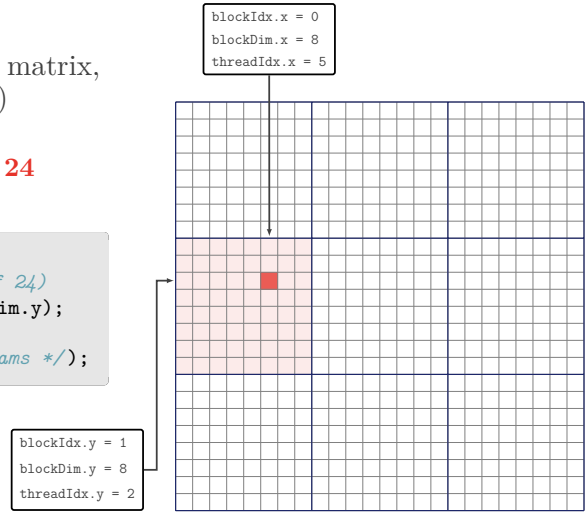
When invoking a CUDA kernel, one must specify:

- The total amount of threads to be created
  - Each thread will execute the same kernel
    - Very different from an OpenMP program, where only a single thread executes the `main` function
- The way threads should be numbered
  - 1D, 2D or 3D: just pick what best fits your algorithm
  - Threads can retrieve their unique id by using:
    - `blockDim.x * blockIdx.x + threadIdx.x`: rank along x dim
    - `blockDim.y * blockIdx.y + threadIdx.y`: rank along y dim
    - `blockDim.z * blockIdx.z + threadIdx.z`: rank along z dim
- Threads need to be grouped in “CUDA blocks”
  - The role of blocks will be discussed later

CUDA Program – Thread Numbering

- Example: kernel working on a 24×24 matrix, with one thread per cell (576 threads)
  - Domain dimensions: **2**
  - Number of threads along each dim: **24**
  - Block size along each dimension: **8**

```
1 dim3 block_dim(8, 8);  // set the block size
2 // set the grid size ('block_dim' multiple of 24)
3 dim3 grid_dim(24 / block_dim.x, 24 / block_dim.y);
4 // invoke kernel on GPU
5 gpu_kernel<<< grid_dim, block_dim >>>(/* params */);
```

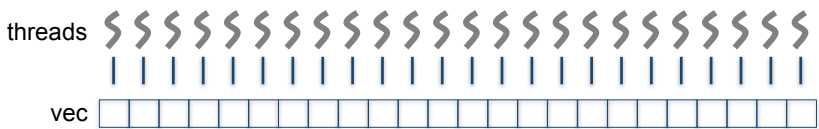


~~Hello World~~ Scale Vector Kernel

- "Hello World" on GPU is not very meaningful...
- Let us try `scal_vec` kernel instead
  - Multiply all the elements of a vector `vec` by a scalar value `k`

```
1 __global__ void scal_vec(float *vec, float k) {
2   size_t id_x = blockDim.x * blockIdx.x + threadIdx.x;
3   vec[id_x] = vec[id_x] * k; // 1 load, 1 mult, 1 store
4 }
```

- Vector `vec` lies in GPU's global memory (hence `vec` buffer)
- The kernel is executed with one thread per vector element
- GPU entry point function (= kernel `scal_vec`) is prefixed by `__global__`



Section 3.1

Practice Time with EasyPAP

EasyPAP – 2D Kernel – Invocation

- Images are `DIM` × `DIM` matrices of `unsigned int`
  - By default, kernels are executed with one thread per element (`GPU_SIZE` = `DIM`)
- Let us observe the kernel invocation side (`kernel/cuda/sample.cu`)

```
1  unsigned sample_compute_cuda(unsigned nb_iter) {
2    // define grid and block sizes
3    dim3 block_dim = {TILE_W, TILE_H, 1};
4    dim3 grid_dim  = {GPU_SIZE_X / block_dim.x, GPU_SIZE_Y / block_dim.y, 1};
5    cudaSetDevice(cuda_device(0)); // select the GPU
6    for (unsigned i = 0; i <= nb_iter; i++) {
7      // submit the kernel execution command to the stream
8      sample_cuda<<<grid_dim, block_dim, 0, cuda_stream(0)>>>(cuda_cur_buffer(0), DIM);
9    }
10   // wait until all commands finished in the stream
11   ret = cudaStreamSynchronize(cuda_stream(0));
12   return 0;
13 }
```

EasyPAP – 2D Sample Kernel – Code

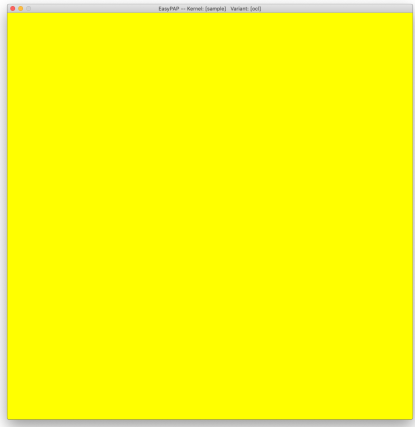
- And now, the kernel itself (`kernel/cuda/sample.cu`)
  - See how each thread retrieves its coordinates (x,y)
  - Note: `pixel(x,y)` of `img` is at offset `y * DIM + x`

```
1 __global__ void sample_cuda(unsigned *img, unsigned DIM)
2 {
3   unsigned x = blockDim.x * blockIdx.x + threadIdx.x; // or 'gpu_get_col()' in EasyPAP
4   unsigned y = blockDim.y * blockIdx.y + threadIdx.y; // or 'gpu_get_row()' in EasyPAP
5   //                  (red, green, blue)
6   unsigned color = rgb(255,   255,    0); // red + green = yellow
7
8   img[y * DIM + x] = color;
9 }
```

- `__global__` decorator specifies that the `sample_cuda` function is a GPU kernel ($\approx$ `main` function in a C program)

EasyPAP – 2D Sample Kernel – Execution

- Let's see how it works on a 256 × 256 image:
  - `./run -k sample -g -s 256`



EasyPAP – 1D Kernel – Invocation

- 1D version of `kernel/cuda/sample.cu`

```
1  unsigned sample_compute_cuda(unsigned nb_iter) {
2    // define grid and block sizes
3    size_t block_dim = TILE_W * TILE_H;
4    size_t grid_dim  = (GPU_SIZE_X * GPU_SIZE_Y) / block_dim;
5    cudaSetDevice(cuda_device(0)); // select the GPU
6    for (unsigned i = 0; i <= nb_iter; i++)
7      // submit the kernel execution command to the stream
8      sample_cuda<<<grid_dim, block_dim, 0, cuda_stream(0)>>>(cuda_cur_buffer(0), DIM);
9    // wait until all commands finished in the stream
10   ret = cudaStreamSynchronize(cuda_stream(0));
11   return 0;
12 }
```

`cuda_device(0)`, `cuda_stream(0)` and `cuda_cur_buffer(0)` are functions defined in EasyPAP. `cuda_device(0)` returns an integer that identify the first GPU, `cuda_stream(0)` returns the first CUDA stream (non-blocking) and `cuda_cur_buffer(0)` returns a data pointer of size DIM × DIM allocated on the GPU global memory (with `cudaMalloc`).

EasyPAP – 1D Sample Kernel – Code

- 1D version of (`kernel/cuda/sample.cu`)

```
1 __global__ void sample_cuda(unsigned *img, unsigned DIM)
2 {
3   unsigned x = blockDim.x * blockIdx.x + threadIdx.x;
4   //                  (red, green, blue)
5   unsigned color = rgb(255,   255,    0); // red + green = yellow
6
7   img[x] = color;
8 }
```

The output is identical ;-).

EasyPAP – 2D Sample Kernel – Code

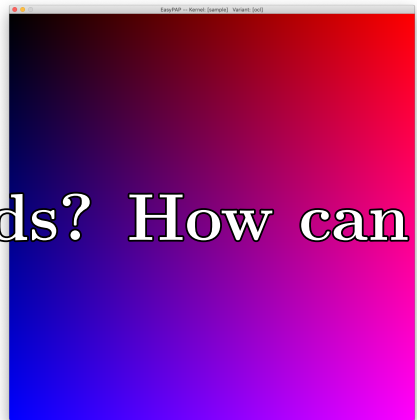
- Back to our 2D version
  - — Let us now introduce coordinate-sensitive colors
  - — To check if `x` and `y` are what we think...

```
__global__ void sample_cuda(unsigned *img, unsigned DIM)
{
  unsigned x = blockDim.x * blockIdx.x + threadIdx.x; // or 'gpu_get_col()' in EasyPAP
  unsigned y = blockDim.y * blockIdx.y + threadIdx.y; // or 'gpu_get_row()' in EasyPAP

  uint8_t red = x & 255; // the greater x, the more red we use
  uint8_t blue = y & 255; // the greater y, the more blue we use
  unsigned color = rgb(red, 0, blue);

  img[y * DIM + x] = color;
}
```

By the way: we use (... & 255) in case the kernel is executed on images larger than 256×256...

EasyPAP – 2D Sample Kernel – Execution

- We run the program the same way (no need to type “make”)
  - `./run -k sample -g -s 4096`



4096^2 threads? How can that be?!?

EasyPAP – 2D Sample Kernel – Discussion

- 16 millions of threads executing the sample kernel? Seriously?
  - — Yes, and it proved to work!
- How can that be?
  - — No existing GPU can manage 16M hardware threads
    - Tesla V100: 80 SM $\times$ 2048 $\approx$ 160K threads
    - At least, not simultaneously!
- Threads are not alive at the same time!
  - — They are executed in batches of thousands
  - — Once a thread terminates, a new one is created
    - Remember: threads creation is (almost) free
  - — Consequently: we must forget global synchronizations (barriers)

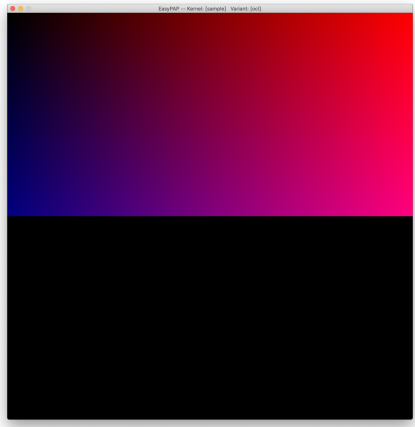
EasyPAP – 2D Kernel – Invocation

- Back to the invocation side (`kernel/cuda/sample.cu`)
  - Let us create only `DIM/2` threads along y
  - `GPU_SIZE_X` = `GPU_SIZE_Y` = `DIM`

```
1  unsigned sample_compute_cuda(unsigned nb_iter) {
2    // define grid and block sizes
3    dim3 block_dim = {TILE_W, TILE_H, 1};
4    dim3 grid_dim  = {GPU_SIZE_X / block_dim.x, (GPU_SIZE_Y / 2) / block_dim.y, 1};
5    cudaSetDevice(cuda_device(0)); // select the GPU
6    for (unsigned i = 0; i <= nb_iter; i++) {
7      // submit the kernel execution command to the stream
8      sample_cuda<<<grid_dim, block_dim, 0, cuda_stream(0)>>>(cuda_cur_buffer(0), DIM);
9    }
10   // wait until all commands finished in the stream
11   ret = cudaStreamSynchronize(cuda_stream(0));
12   return 0;
13 }
```

EasyPAP – 2D Sample Kernel – Execution

- Our kernel does not handle the whole image any more...
  - `./run -k sample -g -s 256`



EasyPAP – 2D Sample Kernel – Code

- Let's fix our kernel to paint the whole image again
  - — Each thread needs to compute 2 pixels

```
__global__ void sample_cuda(unsigned *img, unsigned DIM) {
  unsigned x = blockDim.x * blockIdx.x + threadIdx.x;
  unsigned y = blockDim.y * blockIdx.y + threadIdx.y;

  uint8_t red = x & 255; // the greater x, the more red we use
  uint8_t blue = y & 255; // the greater y, the more blue we use
  unsigned color = rgb(red, 0, blue);
  img[y * DIM + x] = color; // first pixel (x,y)

  // now address the lower half of image
  size_t y2 = y + gridDim.y * blockDim.y; // y2 = y + 128 in our example
  uchar blue2 = y2 & 255; color = rgb(red, 0, blue2);
  img[y2 * DIM + x] = color; // second pixel (x,y2)
}
```

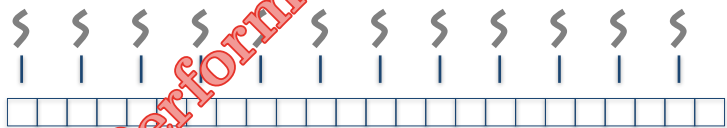
Table of Contents

4 Threads and Global Memory Access

Memory Concerns – Scal Vec – Version 2

- Coming back to our `scal_vec` kernel
  - Same config, except that we spawn size-of-vector/2 CUDA threads

```
1 __global__ void scal_vec_v2(float *vec, float k)
2 {
3   size_t index = blockDim.x * blockIdx.x + threadIdx.x; // thread id (1D)
4
5   vec[index * 2 +0] = vec[index * 2 +0] * k; // 1 load, 1 mult, 1 store
6   vec[index * 2 +1] = vec[index * 2 +1] * k; // 1 load, 1 mult, 1 store
7 }
```



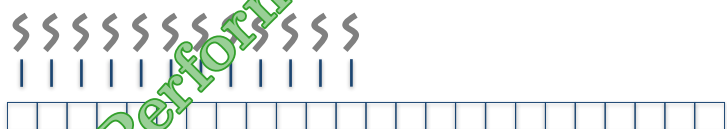
Performance is weak!!

Memory Concerns – Scal Vec – Version 3

4 Threads and Global Memory Access

- Coming back to our `scal_vec` kernel
  - Same config, except that we spawn size-of-vector / 2 CUDA threads

```
1 __global__ void scal_vec_v3(float *vec, float k)
2 {
3   size_t index = blockDim.x * blockIdx.x + threadIdx.x; // thread id (1D)
4   size_t size = gridDim.x * blockDim.x; // # of threads
5
6   vec[index       ] = vec[index       ] * k; // 1 load, 1 mult, 1 store
7   vec[index + size] = vec[index + size] * k; // 1 load, 1 mult, 1 store
8 }
```



Performance is good!!

Memory Access Coalescing

- To exploit full GDDR bandwidth, Nvidia GPUs aggressively try to coalesce contiguous memory accesses into larger ones
- Coalescing is performed at the level of half-warps
  - If 16 contiguous threads access aligned, contiguous memory
    - Then only one large (16-width) memory access is performed
    - Otherwise, up to 16 accesses may be needed
- So, coming back to our previous example
  - `vec[N + blockDim.x * blockIdx.x + threadIdx.x]` is OK
    - ➔ Contiguous threads access contiguous data
  - `vec[2 * (blockDim.x * blockIdx.x + threadIdx.x)]` is NOT OK
    - ➔ Contiguous threads access scattered data

Memory Access Coalescing – Sample Example

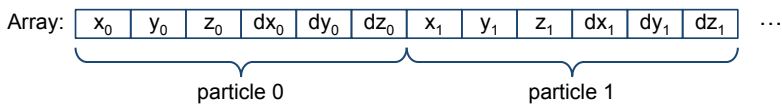
- What if we switch x and y in our `sample` kernel?
  - The output is still correct, but...
  - Performance becomes very weak!
  - Half-warps are contiguous along the x axis
  - But they access vertical columns of data

```
1 __global__ void sample_cuda(unsigned *img)
2 {
3   size_t x = blockDim.x * blockIdx.x + threadIdx.x;
4   size_t y = blockDim.y * blockIdx.y + threadIdx.y;
5
6   unsigned color = rgb (255, 255, 0); // red + green = yellow
7
8   img [x * DIM + y] = color;
9 }
```

Memory Access – Data Layout – Part 1

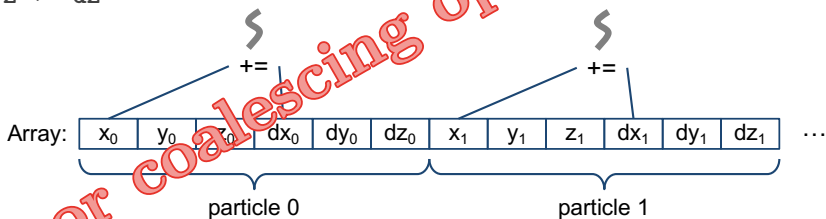
- Example: Moving N particles in a 3D domain
  - Each particle has a position (x, y, z) and a speed vector (d_x, d_y, d_z)
  - We typically use an Array of Structures (aka AoS)
  - Good for cache, isn't it?

```
1 Struct {
2   float x, y, z;    // position
3   float dx, dy, dz; // speed
4 } Particles [N];
```



Memory Access – Data Layout – Part 2

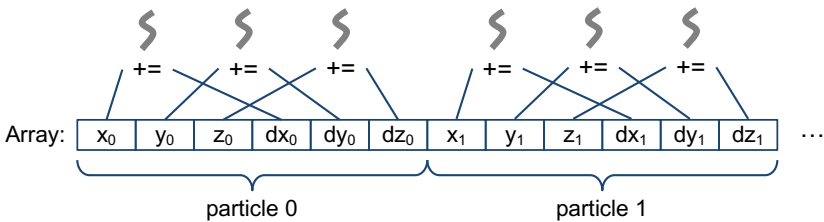
- Moving particles on a GPU
  - One thread per particle
  - `x += dx`
  - `y += dy`
  - `z += dz`



Poor coalescing opportunities!

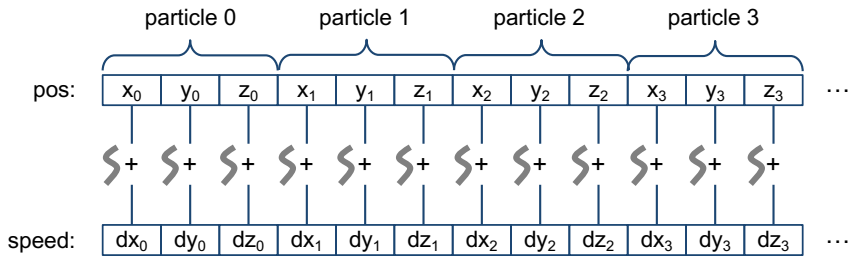
Memory Access – Data Layout – Part 3

- Moving particles on a GPU
  - Would 3 threads per particle help?
  - More parallelism
  - Missed opportunities of coalescing



Memory Access – Data Layout – Part 4

- Moving particles on a GPU
  - Life would be easier if positions and speeds were separated!
  - Moving a particle is simply a 1D vector addition
  - Very efficient on a GPU
  - Each thread blindly adds two scalars
    - No time to think "I'm curious: is that a x that I'm about to modify?"



Memory Access – Data Layout – Part 5

- What if we need to compute distances between particles?

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$$

  — Say for each particle i, we must compute

$$\sum_{0 \le j < N}^{j \neq i} f(d_{ij})$$

- We launch one thread per particle (obviously)
  — When threads access x_j (even for consecutive values of j), addresses are not contiguous!

Memory Access – Data Layout – Part 6

- The good solution is to opt for a "Structure of Arrays" (SoA) layout
  - Six arrays: x, y, z, d_x, d_y, d_z

x:	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	⋯
y:	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	⋯
z:	z_0	z_1	z_2	z_3	z_4	z_5	z_6	z_7	z_8	z_9	⋯
dx:	dx_0	dx_1	dx_2	dx_3	dx_4	dx_5	dx_6	dx_7	dx_8	dx_9	⋯
dy:	dy_0	dy_1	dy_2	dy_3	dy_4	dy_5	dy_6	dy_7	dy_8	dy_9	⋯
dz:	dz_0	dz_1	dz_2	dz_3	dz_4	dz_5	dz_6	dz_7	dz_8	dz_9	⋯

- Actually, this layout also makes a lot of sense for CPUs
  - Vectorization-compliant

Memory Access – Data Layout – Part 7

Always possible to organize data in contiguous chunks?

- Let us consider the "matrix transpose" example
  - Two images: `in` and `out`
  - `in[i,j]` goes to `out[j,i]`, or `in[j,i]` goes to `out[i,j]`
  - In either case, half of memory accesses are bad!
- We'll address this problem later...

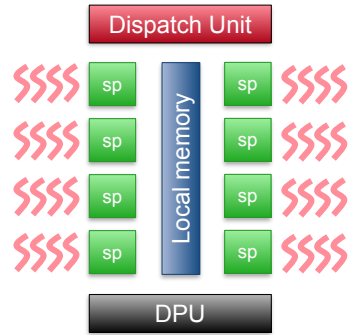
```
1 __global__ void transpose_cuda(const unsigned *in, unsigned *out, size_t DIM)
2 {
3   size_t x = blockDim.x * blockIdx.x + threadIdx.x;
4   size_t y = blockDim.y * blockIdx.y + threadIdx.y;
5
6   out[x * DIM + y] = in[y * DIM + x];
7 }
```

Table of Contents

Threads Divergence – Principle

- Reminder
  - Threads are implicitly grouped in warps of 32 threads
  - All threads of the same warp execute the same instruction at the same logical cycle
  - No divergence!
- No divergence?
  - How to handle conditional code?

```
if(...) ...
(...) ? ... : ...
while(...) ...
```

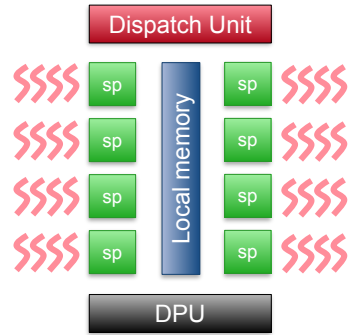


GM80 SM

Threads Divergence – Code and Cycles

- What happens when threads execute a conditional instruction ?
- Threads belonging to the same warp cannot diverge
  - But some of them can “sleep”

```
1 // test if x is even
2 if ((x & 1) == 0) { // all threads execute this at cycle 1
3    do_this(); // half of the threads are working at cycle 2
4 } else {
5    do_that(); // half of the threads are working at cycle 3
6 }
```



GM80 SM

Stripes Kernel – Code

- Impact of thread divergence *wrt* the number of consecutive "buddies" taking the same branch
- The stripes kernel accepts a user parameter
  `./run -l ...  -k stripes -g -c 0`

```
1 unsigned mask = (1 << PARAM);
2 if (x & mask) {
3    out[y * DIM + x] = brighten(in[y * DIM + x]);
4 } else {
5    out[y * DIM + x] = darken(in[y * DIM + x]);
6 }
```

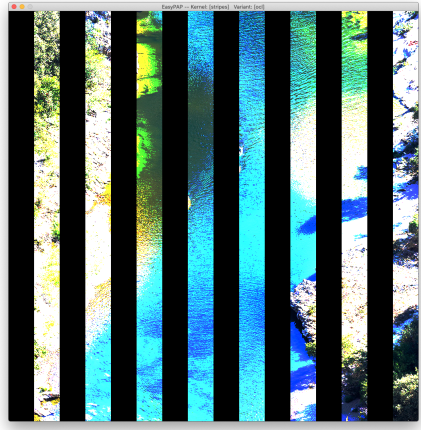
$(x)_{10}$	$(x)_2$	`x & (1 << 0)`
0	00000	00000
1	00001	00001
2	00010	00000
3	00011	00001
4	00100	00000
5	00101	00001
6	00110	00000
7	00111	00001
8	01000	00000
9	01001	00001
10	01010	00000
11	01011	00001
12	01100	00000
13	01101	00001
14	01110	00000
15	01111	00001

Stripes Kernel – Results

5 Threads Divergence

- `PARAM` allow us to control how much consecutive buddies follow the same behavior:
  - The first $2^{\texttt{PARAM}}$ buddies take the same branch, the next $2^{\texttt{PARAM}}$ buddies take the other branch, and so on...

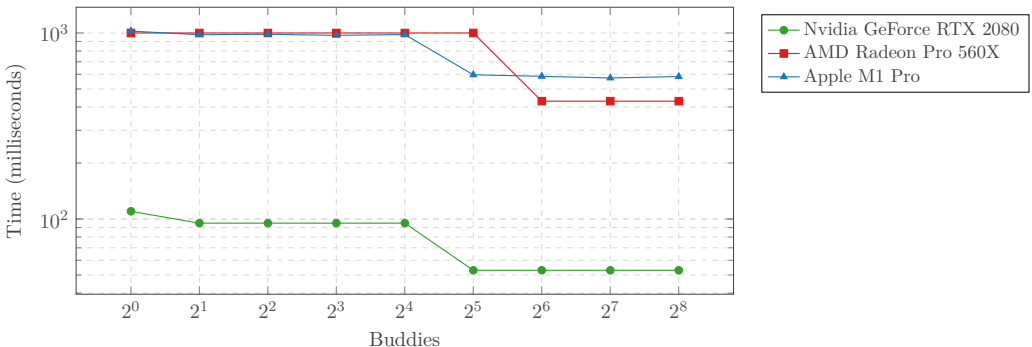
    `./run -l 1024.png -k stripes -g -c 6`



Stripes Kernel – Experiments

5 Threads Divergence

```
./run -k stripes -g -i 1000 -tw 128 -th 2 -n -c <BUDDIES>
```

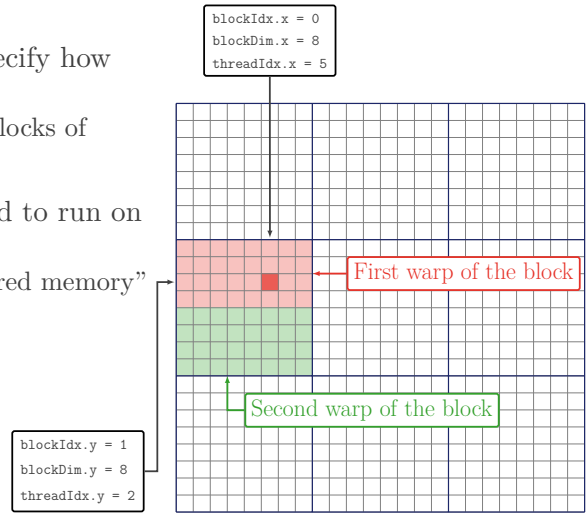


Time for the `stripes` kernel on Nvidia, AMD and M1 Pro GPUs

Table of Contents

Introduction to CUDA Blocks

- When running a kernel, we must specify how threads should be grouped
  - E.g. By default, EasyPAP forms blocks of $16 \times 16 = 256$ threads
- All threads in a block are guaranteed to run on the same SM
  - They can share data through "shared memory"
  - They can synchronize (barriers)
- As a side effect, blocks constrain warp formation
  - E.g. in a 2D 8×8 block
  - Warps spread over four lines of 8 threads

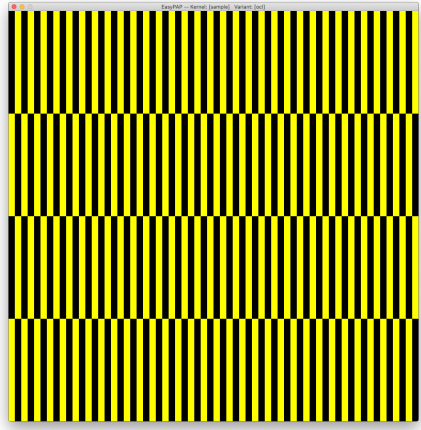


Playing with EasyPAP – Code

6 CUDA Blocks

```
./run -s 256 -k sample -g -tw 4 -th 64
```

```
1  __global__ void sample_cuda(unsigned *img)
2  {
3    size_t x = blockDim.x * blockIdx.x + threadIdx.x;
4    size_t y = blockDim.y * blockIdx.y + threadIdx.y;
5
6    if ((blockIdx.x + blockIdx.y) % 2)
7      img[y * DIM + x] = rgb(255, 255, 0);
8  }
```



Playing with EasyPAP – Experiments

6 CUDA Blocks

On a Nvidia RTX 2070 card:

- `./run -s 256 -k sample -g -tw 16 -th 16 -n -i 1000` ➔ 10.419 ms
- `./run -s 256 -k sample -g -tw 64 -th 04 -n -i 1000` ➔ 11.878 ms
- `./run -s 256 -k sample -g -tw 04 -th 64 -n -i 1000` ➔ 24.382 ms
- `./run -s 256 -k sample -g -tw 04 -th 64 -n -i 1000` ➔ 24.382 ms

Too much uncoalesced memory accesses!

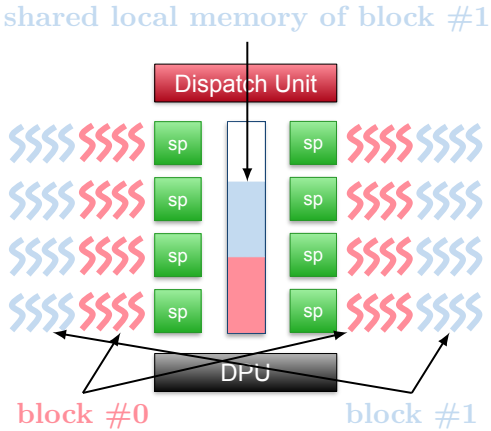
On a Apple Silicon M1 Pro GPU:

- `./run -s 256 -k sample -g -tw 16 -th 16 -n -i 1000` ➔ 12.572 ms
- `./run -s 256 -k sample -g -tw 64 -th 04 -n -i 1000` ➔ 12.624 ms
- `./run -s 256 -k sample -g -tw 04 -th 64 -n -i 1000` ➔ 15.616 ms
- `./run -s 256 -k sample -g -tw 04 -th 64 -n -i 1000` ➔ 15.616 ms

Table of Contents

Introduction to Shared Memory

- Several CUDA blocks can reside on the same streaming multiprocessor
  - Limited by hardware resources
    - Registers
    - Max HW threads per SP
    - Shared Local Memory
- Shared local memory
  - Much faster than global memory
  - Only a few kBytes!
  - No coalescing



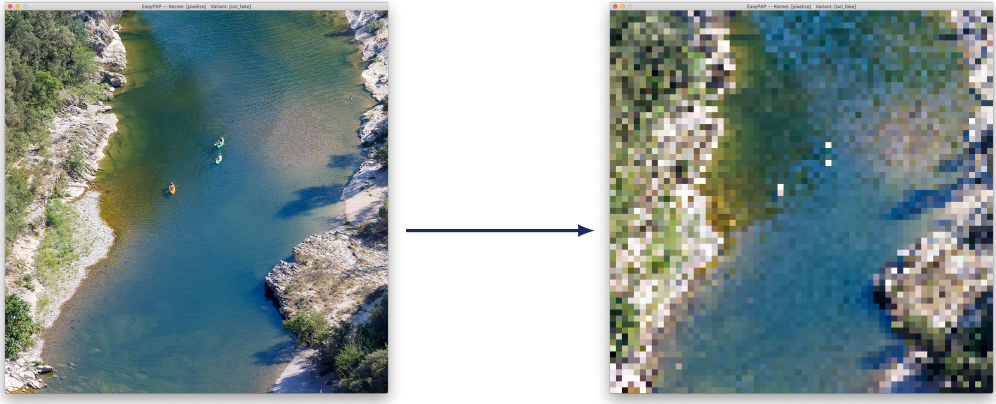
Pixelize Kernel – Code

- Shared memory is declared inside kernels using `__shared__`
- Example with this "oversimplified" `pixelize` kernel
  - Each CUDA block has its private `color` variable

```
1  __global__ void pixelize_cuda(unsigned *img) {
2    size_t xl = threadIdx.x, yl = threadIdx.y;
3    size_t xg = blockDim.x * blockIdx.x + threadIdx.x;
4    size_t yg = blockDim.y * blockIdx.y + threadIdx.y;
5    // declare shared memory (shared by all the threads of a block)
6    __shared__ unsigned color;
7    // thread from the upper left corner sets this shared variable
8    if (xl == 0 && yl == 0)
9      color = img[yg * DIM + xg]; // only one thread per block reads from global memory
10   // synchronize block threads to make sure 'color' has been written in the shmem
11   __syncthreads();
12   // finally, all threads write their pixel to the one contained in 'color'
13   img[yg * DIM + xg] = color;
14 }
```

Pixelize Kernel – Results


```
./run -l images/1024.png -k pixelize -g -tw 16 -th 16
```



Static versus Dynamic Shared Mem. Alloc.

Static shared memory allocation

```
1 __global__ void cuda_kernel(/* kernel params */) {
2   __shared__ unsigned shmem[N]; // 'N' is a define or a static constant
3   /* ... */
4 }
```

Dynamic shared memory allocation

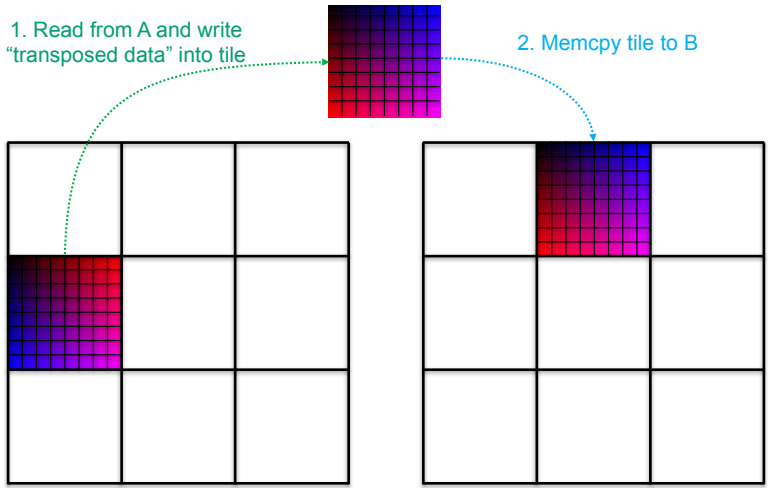
```
1 // invoke kernel from the host, 'n' can be dynamic in the host code
2 cuda_kernel<<< grid_dim, block_dim, n * sizeof(unsigned) >>>(/* kernel params */);
```

```
1 __global__ void cuda_kernel(/* kernel params */) {
2   extern __shared__ unsigned shmem[];
3   /* ... */
4 }
```

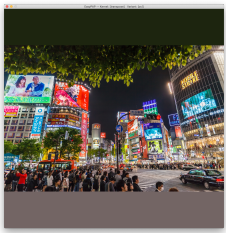
From Pixelize to Transpose Kernel

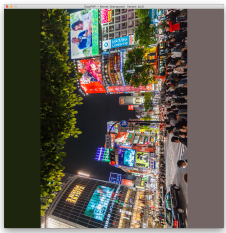
- CUDA blocks can share more than a scalar value
  - E.g. `__shared__ unsigned tile[TILE_H * TILE_W];`
  - Serves as a "cache" in which data is fetched from global memory (this is a ScratchPad Memory (SPM))
  - Contrary to a cache, the developer needs to make the copy explicitly in the local memory
- Let us use such "tiles" to solve our transpose problem!
  - Use shared memory to compute transposed tiles
  - "memcpy" tiles to the right place in global memory
  - Keep global memory accesses contiguous!
  - As usual, we spawn one thread per matrix element

Transpose Kernel – Principle



Transpose Kernel – Code



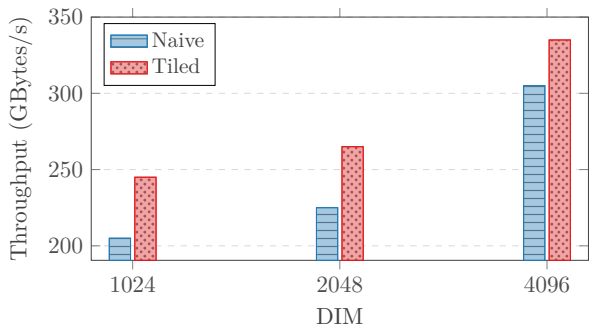


```
./run -k transpose -v cuda_tiled -l shib*.png -r 1 -g -p
```

```
__global__ void transpose_cuda_tiled(
  const uint32_t *in, uint32_t *out, size_t DIM)
{
  extern __shared__ uint32_t tile[/* bDim.x * bDim.y */];
  size_t xl = threadIdx.x, yl = threadIdx.y;
  size_t xg = blockDim.x * blockIdx.x + threadIdx.x;
  size_t yg = blockDim.y * blockIdx.y + threadIdx.y;

  tile[xl * blockDim.x + yl] = in[yg * DIM + xg];

  __syncthreads();

  out[(xg - xl + yl) * DIM + yg - yl + xl]
    = tile[yl * blockDim.x + xl];
}
```

Transpose Kernel – Experiments 1

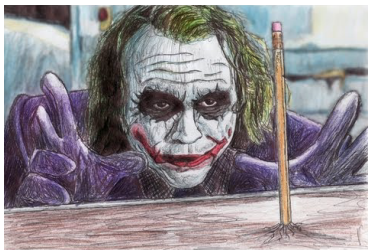

```
./run -g -k transpose -v <cuda_naif|cuda_tiled> -i 1000 -n -s <DIM>
```



Throughput of the `transpose` kernel on Nvidia GeForce GTX 2080.

Transpose Kernel – Magic Trick

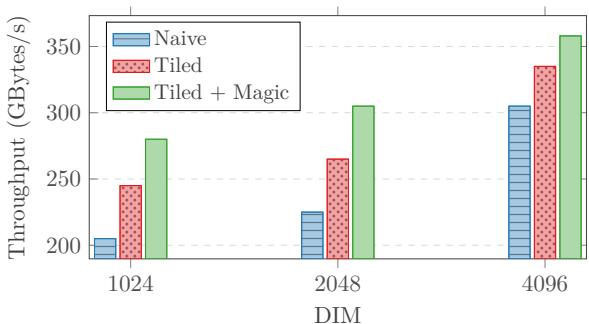
Why the hell do we add this extra column we don't even use?!



```
__global__ void transpose_cuda_tiled(
  const uint32_t *in, uint32_t *out, size_t DIM)
{
  __shared__ uint32_t tile[/* (bDim.x + 1) * bDim.y */];
  size_t xl = threadIdx.x, yl = threadIdx.y;
  size_t xg = blockDim.x * blockIdx.x + threadIdx.x;
  size_t yg = blockDim.y * blockIdx.y + threadIdx.y;

  tile[xl * (blockDim.x + 1) + yl] = in[yg * DIM + xg];

  __syncthreads();

  out[(xg - xl + yl) * DIM + yg - yl + xl]
    = tile[yl * (blockDim.x + 1) + xl];
}
```

Transpose Kernel – Experiments 2

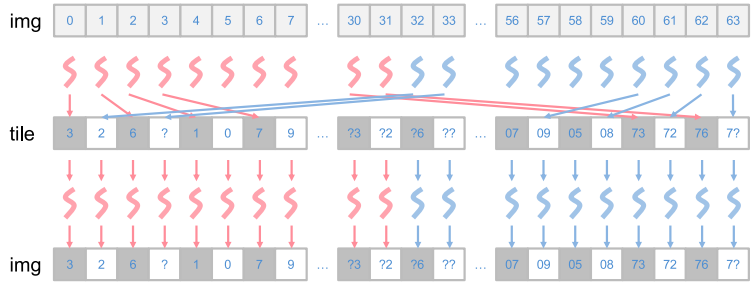

```
./run -g -k transpose -v <cuda_naif|cuda_tiled> -i 1000 -n -s <DIM>
```



Throughput of the `transpose` kernel on Nvidia GeForce GTX 2080

Stripes Kernel – Principle

- Back to `stripes`
  - Assuming `TILE_W` is a multiple of 64...
- Implementation of 1-pixel-width stripes
  - Divergence avoiding & Memory coalescing



Stripes Kernel – Code

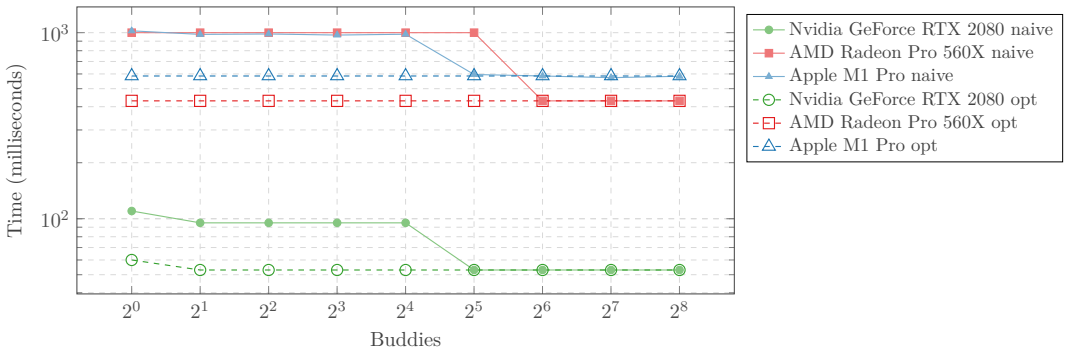
- Implementation of 1-pixel-width stripes
  - Divergence avoiding
  - Memory coalescing
- Do not forget about the synchros!

```
1  extern __shared__ unsigned tile[/* bDim.x*bDim.y */];
2  size_t xl = threadIdx.x, yl = threadIdx.y;
3  size_t xg = blockDim.x * blockIdx.x + threadIdx.x;
4  size_t yg = blockDim.y * blockIdx.y + threadIdx.y;
5  size_t index = 2 * xl;
6  tile[yl * blockDim.x + xl] = in[yg * DIM + xg];
7  __syncthreads();
8  if (index < blockDim.x) { // 1 warp takes the if
9    tile[yl * blockDim.x + index]
10     = darken(tile[yl * blockDim.x + index]);
11 } else { // 1 warp takes the else
12   index += -blockDim.x + 1;
13   tile[yl * blockDim.x + index]
14     = brighten(tile[yl * blockDim.x + index]);
15 }
16 __syncthreads();
17 out[yg * DIM + xg] = tile[yl * blockDim.x + xl];
```

Stripes Kernel – Experiments

7 Shared Memory

```
./run -k stripes -g -i 1000 -tw 128 -th 2 -n -c <BUDDIES>
```



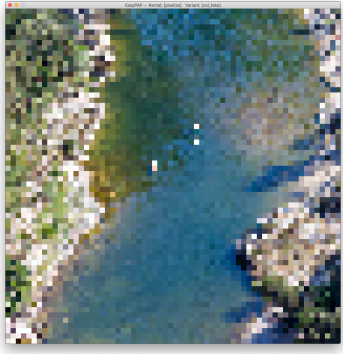
Time for the `stripes` kernel on Nvidia, AMD and M1 Pro GPUs

Table of Contents

Accurate Pixelize Kernel – Principle

8 Reductions

```
./run -l 1024.png -k pixelize -g
```



Simplified version

Expected version

Accurate Pixelize Kernel – Accumulate

8 Reductions

- Adding colors
  - Pixels are stored as unsigned integers (RGBA8888 format)
- Adding two raw pixels may lead to value overflow
  - Convert each 8-bit component to a larger, separate integer
  - CUDA provide “vectors” of 2, 3 or 4 scalar values

```
int4 v;
v.x = 3; ... v.w = 5;
```

```
1  __global__ void pixelize_cuda(unsigned *in, size_t DIM)
2  {
3    extern __shared__ int4 tile[/* bDim.x * bDim.y */];
4
5    size_t xl = threadIdx.x, yl = threadIdx.y;
6    size_t xg = blockDim.x * blockIdx.x + threadIdx.x;
7    size_t yg = blockDim.y * blockIdx.y + threadIdx.y;
8
9    tile[yl * blockDim.x + xl]
10     = color_to_int4(in[yg * DIM + xg]);
11   // ...
12 }
```

Accurate Pixelize Kernel – Reduction

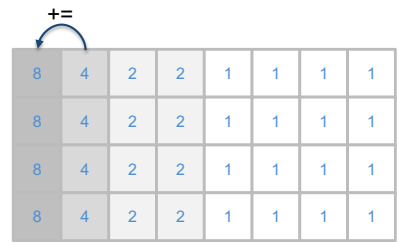
- Reduction
  - We first cache pixels into local memory
  - Then we can perform our 2D reduction inside tile
  - There is one thread per cell
    - To maximize throughput of load operation
- How do we compute the sum of all cells?

```
1  __global__ void pixelize_cuda(unsigned *in, size_t DIM)
2  {
3    extern __shared__ int4 tile[/* bDim.x * bDim.y */];
4
5    size_t xl = threadIdx.x, yl = threadIdx.y;
6    size_t xg = blockDim.x * blockIdx.x + threadIdx.x;
7    size_t yg = blockDim.y * blockIdx.y + threadIdx.y;
8
9    tile[yl * blockDim.x + xl]
10     = color_to_int4(in[yg * DIM + xg]);
11   __syncthreads();
12   // ...
13 }
```

Accurate Pixelize Kernel – Horizontal Reduc.

8 Reductions

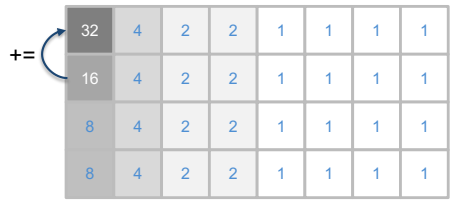
- Let's consider tiles of 8×4 cells
- How do we compute the sum of all cells?
  - — Well, we could perform a first wave of 4×4 additions
    - 4 additions per row
    - Half of threads do not work
- Next step
  - — Second wave of 2×4 additions
    - 2 additions per row
    - Only 1/4 of threads participate
- Next step
  - — Second wave of 1×4 additions
    - 1 additions per row
    - Only 1/8 of threads participate




```
1 if (xl < 1)
2   tile[yl * bDim.x + xl]
3     += tile[yl * bDim.x + (xl + 1)];
```

Accurate Pixelize Kernel – Vertical Reduction

- Now what?
- We sum up the cells vertically, but only for the first column
  - — Avoid wasting local memory bandwidth
  - — Only two threads participate
  - — Last step: one thread participates




```
1 if (xl == 0) {
2   if (yl < 1)
3     tile[yl * bDim.x + 0]
4       += tile[(yl + 1) * bDim.x + 0];
5 }
```

Accurate Pixelize Kernel – Code

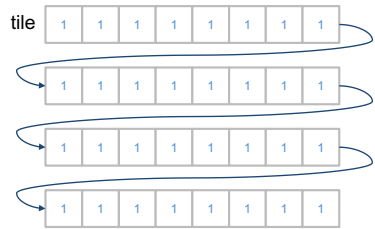

```
__global__ void pixelize_cuda_accurate(unsigned *in, size_t DIM) {
  extern __shared__ int4 tile[/* blockDim.x * blockDim.y */];
  size_t xl = threadIdx.x, yl = threadIdx.y;
  size_t xg = blockDim.x * blockIdx.x + threadIdx.x, yg = blockDim.y * blockIdx.y + threadIdx.y;

  tile[yl * blockDim.x + xl] = color_to_int4(in[yg * DIM + xg]);

  // Averaging each line
  for (int d = blockDim.x >> 1; d > 0; d >>= 1) {
    __syncthreads();
    if (xl < d) // '+' operator is not avaiblable on int4 type, need to perform 'add' element-wise
      tile[yl * blockDim.x + xl] += tile[yl * blockDim.x + (xl + d)]; // t[].x +=, t[].y +=, t[].z +=, t[].w +=
  }
  // Averaging first column only
  for (int d = blockDim.y >> 1; d > 0; d >>= 1) {
    __syncthreads();
    if (xl == 0 && yl < d) // '+' operator is not avaiblable on int4 type, need to perform 'add' element-wise
      tile[yl * blockDim.x + xl] += tile[(yl + d) * blockDim.x + xl]; // t[].x +=, t[].y +=, t[].z +=, t[].w +=
  }

  __syncthreads();
  in[yg * DIM + xg] = int4_to_color(tile[0] / (int4)(blockDim.x * blockDim.y));
}
```

Accurate Pixelize Kernel – Alternative

- A simpler solution is to consider the tile as a 1D array!

```
1 __shared__ int4 tile[/* bDim.x * bDim.y */];
2 size_t xl = threadIdx.y * bDim.x + threadIdx.x;
```

- Perform a simple 1D-reduction of all values in tile



From 2D to 1D

Accurate Pixelize Kernel – Code

8 Reductions

```
__global__ void pixelize_cuda_accurate(unsigned *in, size_t DIM)
{
  extern __shared__ int4 tile[/* blockDim.x * blockDim.y */];
  size_t xl = threadIdx.y * blockDim.x  + threadIdx.x;
  size_t xg = blockDim.x * blockIdx.x + threadIdx.x;
  size_t yg = blockDim.y * blockIdx.y + threadIdx.y;

  tile[xl] = color_to_int4(in[yg * DIM + xg]);

  for (int d = (blockDim.x * blockDim.y) >> 1; d > 0; d >>= 1) {
    __syncthreads();
    if (xl < d) // '+' operator is not avaiblable on int4 type, need to perform 'add' element-wise
      tile[xl] += tile[xl + d]; // tile[].x +=, tile[].y +=, tile[].z +=, tile[].w +=
  }
  __syncthreads();

  in[yg * DIM + xg] = int4_to_color(tile[0] / (int4) (blockDim.x * blockDim.y));
}
```

Final Notes about Reductions

- Block-wide reductions can be speedup with specific intrinsics
  - `__shfl_sync`: per-warp instrinsic function that does not require shared memory usage...
- For reduction on large data sets (> block max size), multi-pass kernels must be used
  - No accelerator-wide barrier
  - Barrier between successive kernels
    - Each kernel performs separate per-block reductions, and write results in memory
    - Loop until #elements $\leq$ block size

Table of Contents

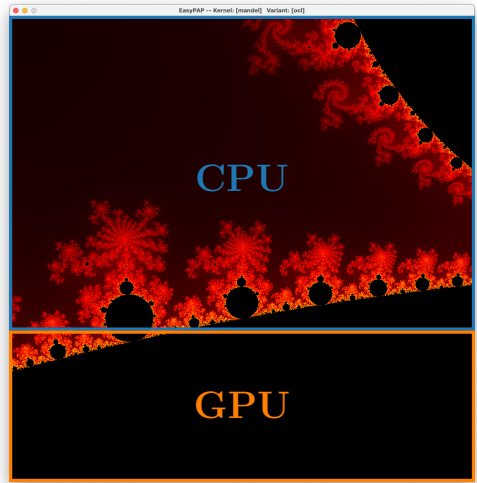
Example with the Mandelbrot Kernel

- Mandelbrot is a compute-bound kernel
  - No memory access challenge
  - No data reuse
  - There is intra-warp divergence (as in the SIMD version...)
- Kernel implementation is straightforward...
- Code is similar to the sequential one!

```
__global__ void mandel_cuda(unsigned *img, size_t DIM,
  float leftX, float xstep, float topY, float ystep,
  unsigned MAX_ITERATIONS)
{
  size_t i = blockDim.x * blockIdx.x + threadIdx.x;
  size_t j = blockDim.y * blockIdx.y + threadIdx.y;

  float xc = leftX + xstep * j;
  float yc = topY - ystep * i;
  float x = 0.0, y = 0.0; // Z = X + i*Y

  unsigned iter;
  for (iter = 0; iter < MAX_ITERATIONS; iter++) {
    float x2 = x * x, y2 = y * y;
    if (x2 + y2 > 4.0) // Stop iterations when |Z| > 2
      break;
    float twoxy = (float)2.0 * x * y;
    x = x2 - y2 + xc;
    y = twoxy + yc;
  }

  img[i * DIM + j] = (iter < MAX_ITERATIONS) ?
    /* then */ mandel_iter2color(iter) :
    /* else */ 0x000000FF; // black
}
```

Mandelbrot Hybrid CPU+GPU Version

- Implementing a CPU+GPU Mandelbrot should be a no-brainer
  - No data exchange needed between iterations!
- Fixed partitioning
  - CPU takes `n` tile rows
  - GPU takes `NB_TILES_Y - n` tile rows
- Go and try with EasyPAP!



Q&A

Thank you for listening!
Do you have any questions?