GPU Programming with OpenCL

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:

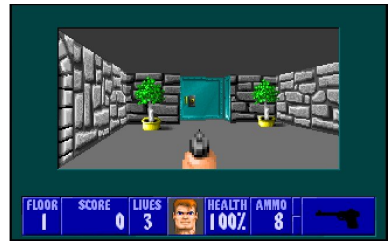
- https://gforgeron.gitlab.io/pap/
- https://raymond-namyst.emi.u-bordeaux.fr/ens/pap/PAP-GPU.pdf

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

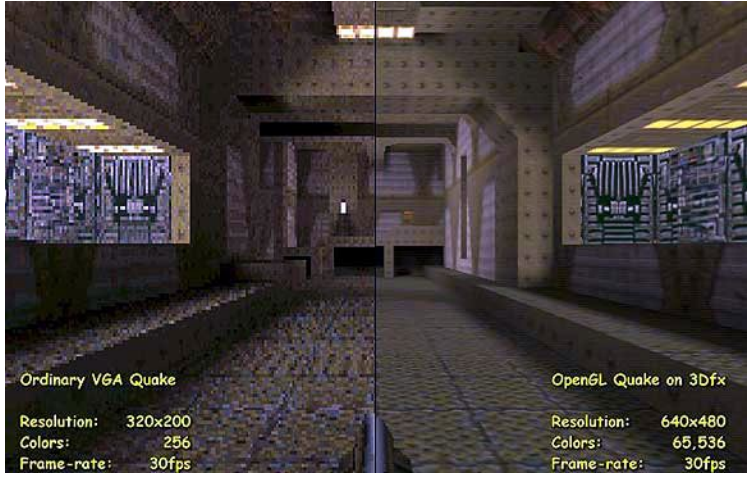
SEGA Virtua Fighter Remix for Diamond Edge3D (NV1)

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)

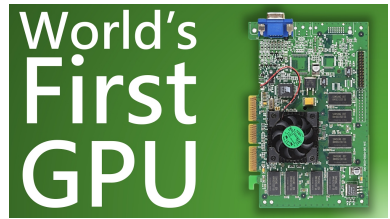
Tomb Raider (DOS, 1996)

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

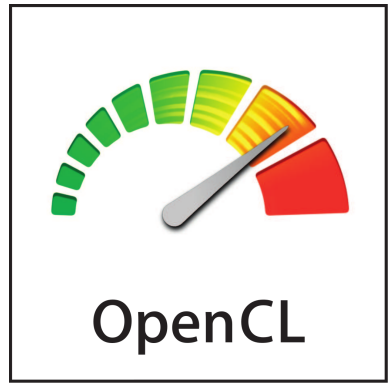
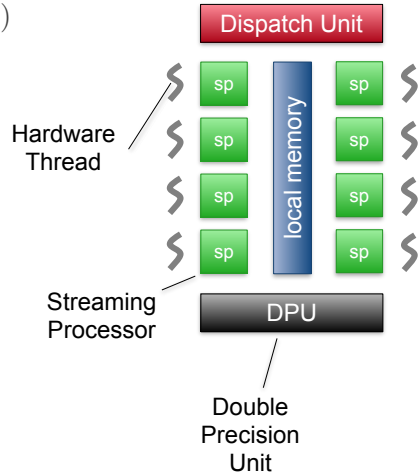


Table of Contents

2 Execution Model

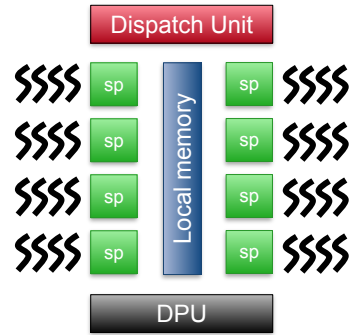
Introduction to the Nvidia Execution Model

- 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)



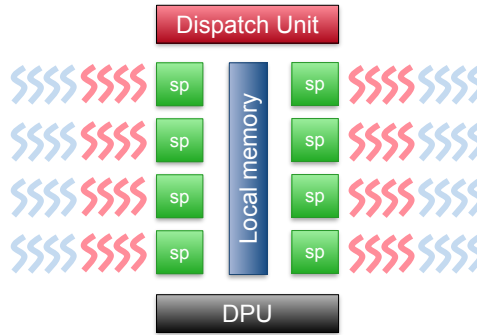
Single Instruction Multiple ~~Data~~ Threads

- 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!



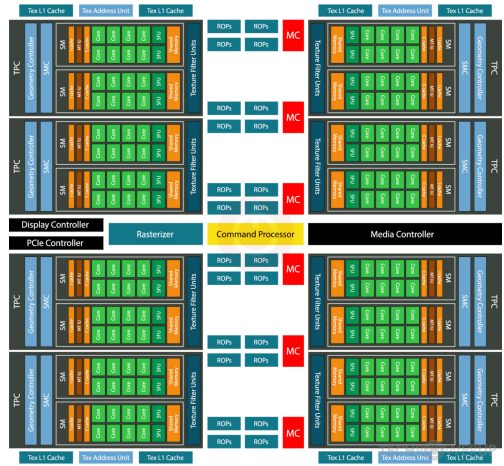
Warps and Half-warps

- 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



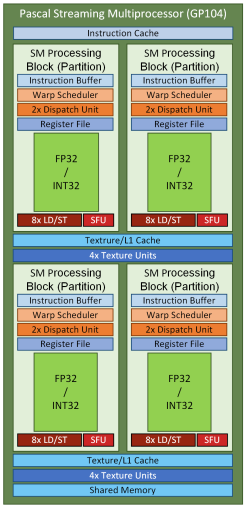
The Whole Picture

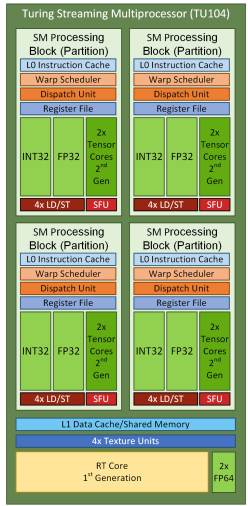
- GPU = set of Streaming Multiprocessors sharing a global memory
- Nvidia GeForce 8800 GTX ($\approx$ Tesla C870)
  - 16 SM
  - $8 \times 16 = 128$ CUDA cores
  - 128 threads max per processor = 16,384 threads!
- Not exactly the usual meaning of “thread”...
  - Data-parallelism
  - Regular access patterns

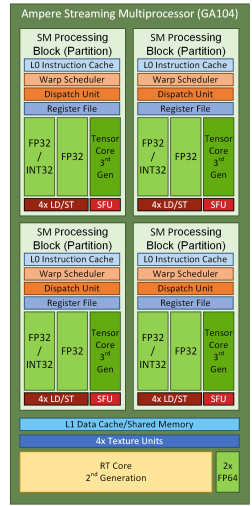


Modern Streaming Multiprocessors

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

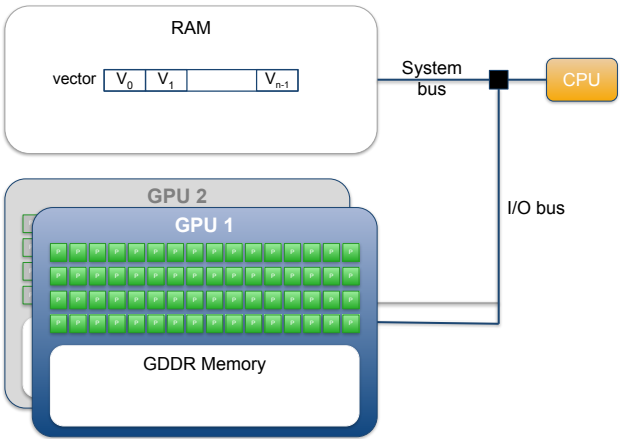
3 OpenCL Programming Environment

Presentation

- OpenCL is both a set of library routines and a programming language
- Library routines categories:
  - Hardware discovery
  - Device (e.g. GPU) selection
  - On-device memory management
  - Memory transfers
  - Program compilation
  - Program launch

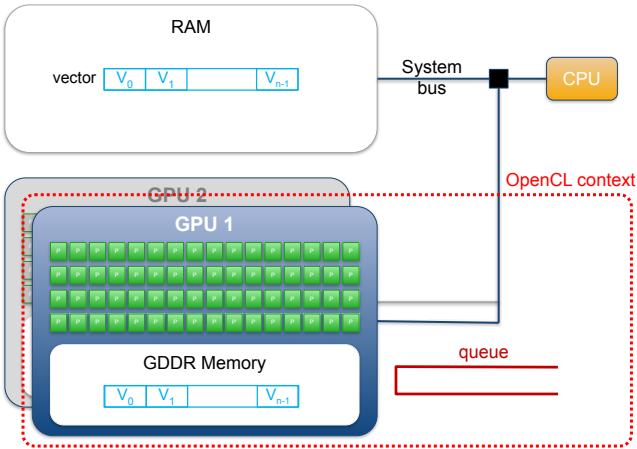
- OpenCL language
  - C language + a few keywords
  - Code is compiled, sent to device, and executed
  - Code entry points are named “kernels”
  - Kernel ≈ main function of a GPU program
    - Can be invoked from CPU side
  - Notable differences:
    - Kernels are executed in parallel by many threads

The Big Picture



- CPU, RAM, BUS and GPUs
- How to modify our vector on GPU 1?

The Big Picture – Main Steps



1. Setup OpenCL context and work queue
2. Allocate memory on GPU
3. Send data to GPU
4. Compile OpenCL “kernel”
5. Execute kernel on GPU
6. Retrieve data back to RAM
7. Enjoy your vector

Typical Workflow of a Simple OpenCL Program

- An OpenCL program typically follows these steps:
  1. Configure an OpenCL “queue” which will serve as a mean to send orders to the target GPU
  2. Allocate memory on GPU side
  3. Transfer (copy) input data from RAM to GPU memory
  4. Compile kernel for the target GPU architecture
  5. Execute kernel on GPU (detailed later)
  6. Retrieve output data (copy) from GPU memory to RAM
  7. Use the results on the CPU side!
- Let us see this in more details in the next slides

OpenCL API – 1. Context and Cmd Queue


```
1  // get platform and device information
2  cl_uint num_platforms;
3  // set up the platform
4  clGetPlatformIDs(0, NULL, &num_platforms);
5  cl_platform_id *platforms = malloc(sizeof(cl_platform_id) * num_platforms);
6  clGetPlatformIDs(num_platforms, platforms, NULL);
7  // get the devices list and choose the device you want to run on
8  cl_uint num_devices;
9  clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_GPU, 0, NULL, &num_devices);
10 cl_device_id *devices_list = malloc(sizeof(cl_device_id) * num_devices);
11 clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_GPU, num_devices, devices_list, NULL);
12 // create one OpenCL context for each device in the platform
13 cl_context context = clCreateContext(NULL, num_devices, devices_list, NULL, NULL, NULL);
14 // create a command queue
15 cl_command_queue command_queue = clCreateCommandQueue(context, devices_list[0], 0, NULL);
```

OpenCL API – 2. Memory Buffers Allocation


```
1  unsigned int n_elements = 1000;
2  // create a read only buffer (read only on the GPU but the CPU can write into it)
3  cl_mem ro_gpu_buff = clCreateBuffer(context, CL_MEM_READ_ONLY,
4                                      n_elements * sizeof(cl_float), NULL, NULL);
5
6  // create a write only buffer (write only on the GPU but the CPU can read into it)
7  cl_mem wo_gpu_buff = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
8                                      n_elements * sizeof(cl_float), NULL, NULL);
9
10  // create a read and write buffer
11 cl_mem rw_gpu_buff = clCreateBuffer(context, CL_MEM_READ_WRITE,
12                                     n_elements * sizeof(cl_float), NULL, NULL);
```

OpenCL API – 3. Write from Host to Device


```
// allocate and initialize a buffer on the host (CPU) memory space
unsigned int n_elements = 1000;
float* cpu_buff = malloc(sizeof(float) * n_elements); // alloc

for (unsigned int i = 0; i < n_elements; i++)
  cpu_buff[i] = i; // init

// copy buffer from host (CPU) to device (GPU)
clEnqueueWriteBuffer(command_queue, ro_gpu_buff, CL_TRUE, 0,
                     n_elements * sizeof(cl_float),
                     cpu_buff, 0, NULL, NULL);
// 'CL_TRUE' ensures that after the 'clEnqueueWriteBuffer' call, the data have
// been copied on the GPU (call is synchronous)
```

OpenCL API – 4. Compile a GPU Kernel


```
1  // read kernels source code from the 'kernels.cl' file
2  FILE *kernels_file = fopen("kernels.cl", "r");
3  fseek(kernels_file, 0, SEEK_END);
4  size_t file_size = ftell(kernels_file);
5  fseek(kernels_file, 0, SEEK_SET);
6
7  char *kernels_source = malloc((file_size + 1) * sizeof(char));
8  fread(kernels_source, sizeof(char), file_size, kernels_file);
9  fclose(kernels_file);
10 kernels_source[file_size] = '\0';
11
12 // create a program from the kernels source code
13 cl_program program = clCreateProgramWithSource(context, 1, &kernels_source, NULL, NULL);
14 // build/compile the program (can contain multiple kernels)
15 clBuildProgram(program, 1, &(devices_list[0]), NULL, NULL, NULL);
16 // create the OpenCL kernel ('k1' has to exist in the 'kernels.cl' file)
17 cl_kernel kernel1 = clCreateKernel(program, "k1", NULL);
```

OpenCL API – 5. Execute a GPU Kernel


```
1  // let us suppose that the 'k1' prototype is the following in the 'kernel.cl' file
2  __kernel void k1(__global const float *in_buff, __global float *out_buff, unsigned n);
```

```
1  // set the kernel parameters
2  clSetKernelArg(kernel1, 0, sizeof(ro_gpu_buff), &ro_gpu_buff);
3  clSetKernelArg(kernel1, 1, sizeof(wo_gpu_buff), &wo_gpu_buff);
4  clSetKernelArg(kernel1, 2, sizeof(unsigned int), &n_elements);
5
6  // execute the OpenCL kernel on the GPU
7  const cl_uint work_dim = ???; // we will see this later, patience young padawan ;-)
8  const size_t global_work_size = ???; // we will see this later, patience young padawan ;-)
9  const size_t local_work_size = ???; // we will see this later, patience young padawan ;-)
10 clEnqueueNDRangeKernel(command_queue, kernel1, work_dim, NULL,
11                        &global_work_size, &local_work_size, 0, NULL, NULL);
12 // here we have no guarantee that the execution is finished or even started!
13 // 'clEnqueueNDRangeKernel' is an async call that just send a cmd to the GPU queue
```

OpenCL API – 6. Read from Device to Host


```
1  // copy buffer from device (GPU) to host (CPU)
2  clEnqueueReadBuffer(command_queue, wo_gpu_buff, CL_TRUE, 0,
3                      n_elements * sizeof(cl_float), (cl_float *)cpu_buff, 0, NULL, NULL);
4  // 'CL_TRUE' ensures that after the 'clEnqueueReadBuffer' call, the data have
5  // been copied on the CPU (and that the kernel execution is finished on the CPU)
6  // this is a synchronous call
7
8  // here you can use 'cpu_buff' as usual on the host (CPU) side
9  for (unsigned int i = 0; i < n_elements; i++) {
10   printf("cpu_buff[%u] = %f\n", i, cpu_buff[i]);
11 }
```

OpenCL API – Summary

- All the previous slides present C code executed by the CPU
  - The CPU (host) controls the GPU (device)
  - OpenCL API is a standard C library
- We did not see
  - Some mysterious `clEnqueueNDRangeKernel` arguments
  - The code executed on GPU ➔ OpenCL program

OpenCL Program

- Contrary to the OpenCL API, OpenCL program target GPUs
- It is an extension of the C language
  - New keywords: `__kernel`, `__global`, `__local`, etc.
  - Intrinsic (predefined) functions (`get_global_id`, `get_local_id`, etc.)
- Although not required, it is a good idea to store OpenCL programs in `*.cl` disk files
- An OpenCL 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 an OpenCL program can expose multiple kernels

OpenCL Program – Kernel Invocation

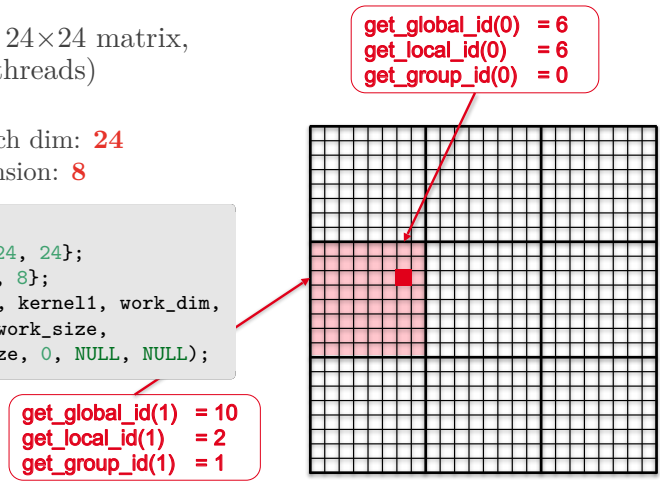
When invoking an OpenCL kernel, one must specify:

- 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:
    - `get_global_id(0)`: rank along x axis
    - `get_global_id(1)`: rank along y axis (if dim $>$ 1D)
    - `get_global_id(2)`: rank along z axis (if dim $=$ 3D)
- Threads should be grouped in so-called OpenCL workgroups
  - The role of workgroups will be discussed later

OpenCL 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**
  - Group size along each dimension: **8**

```
1 const cl_uint work_dim = 2;
2 const size_t global_work_size[2] = {24, 24};
3 const size_t local_work_size[2] = {8, 8};
4 clEnqueueNDRangeKernel(command_queue, kernel1, work_dim,
5                        NULL, global_work_size,
6                        local_work_size, 0, NULL, NULL);
```

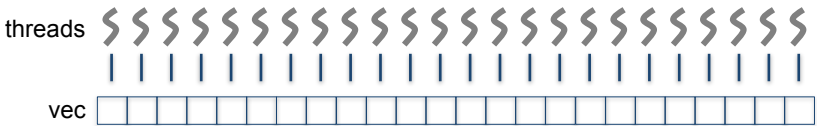


~~Hello World~~ Scale Vector Kernel

- "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 __kernel void scal_vec(__global float *vec, float k) {
2   size_t index = get_global_id(0); // thread id (1D)
3   vec[index] = vec[index] * k; // 1 load, 1 mult, 1 store
4 }
```

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



Section 3.1

Practice Time with EasyPAP

EasyPAP – 2D Kernel – Invocation

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

```
1  unsigned sample_compute_ocl(unsigned nb_iter) {
2    size_t global[2] = {GPU_SIZE_X, GPU_SIZE_Y}; // global domain size for our calculation
3    size_t local [2] = {TILE_W,     TILE_H    }; // local  domain size for our calculation
4    for (unsigned it = 1; it <= nb_iter; it++) {
5      // set kernel arguments
6      clSetKernelArg(compute_kernel, 0, sizeof (cl_mem), &cur_buffer);
7      // submit the kernel execution command to the queue
8      clEnqueueNDRangeKernel(queue, compute_kernel, 2, NULL, global, local, 0, NULL, NULL);
9    }
10   clFinish (queue); // wait until all commands finished in the queue
11   return 0;
12 }
```

EasyPAP – 2D Sample Kernel – Code

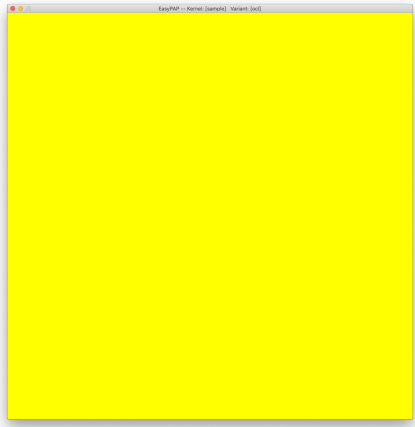
- And now, the kernel itself (`kernel/ocl/sample.cl`)
  - See how each thread retrieves its coordinates (x,y)
  - Note: `pixel(x,y)` of `img` is at offset `y * DIM + x`

```
1 __kernel void sample_ocl(__global unsigned *img)
2 {
3     size_t x = get_global_id(0);
4     size_t y = get_global_id(1);
5     //                    (red, green, blue)
6     unsigned color = rgb (255,   255,     0); // red + green = yellow
7
8     img[y * DIM + x] = color;
9 }
```

By the way: OpenCL kernels are compiled with `-DDIM=<Image size>` (that's why we can use `DIM` here).

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

- Note: we could use 1D numbering as well
  - Create `DIM` × `DIM` threads
- 1D version of `kernel/c/sample.c`

```
1  unsigned sample_compute_ocl(unsigned nb_iter) {
2    size_t global[1] = {GPU_SIZE_X * GPU_SIZE_Y}; // global domain size for our calculation
3    size_t local [1] = {TILE_W * TILE_H}; // local  domain size for our calculation
4    for (unsigned it = 1; it <= nb_iter; it++) {
5      // set kernel arguments
6      clSetKernelArg(compute_kernel, 0, sizeof (cl_mem), &cur_buffer);
7      // submit the kernel execution command to the queue
8      clEnqueueNDRangeKernel(queue, compute_kernel, 1, NULL, global, local, 0, NULL, NULL);
9    }
10   clFinish (queue); // wait until all commands finished in the queue
11   return 0;
12 }
```

EasyPAP – 1D Sample Kernel – Code

- 1D version of (`kernel/ocl/sample.cl`)

```
1 __kernel void sample_ocl(__global unsigned *img)
2 {
3   size_t index = get_global_id(0);
4   //                    (red, green, blue)
5   unsigned color = rgb (255,   255,    0); // red + green = yellow
6
7   img[index] = color;
8 }
```

The output is identical ;-).

EasyPAP – 2D Sample Kernel – Code

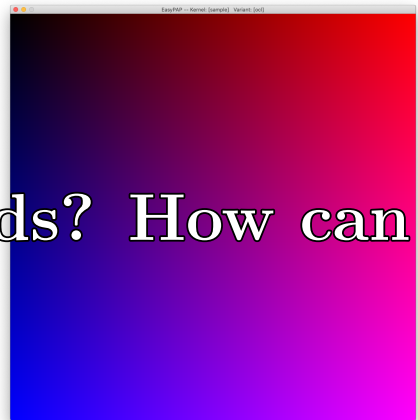
- Back to our 2D version
  - Let us now introduce coordinate-sensitive colors
  - To check if `x` and `y` are what we think...

```
1  __kernel void sample_ocl(__global unsigned *img)
2  {
3  size_t x = get_global_id(0);
4  size_t y = get_global_id(1);
5
6  uchar red = x & 255; // the greater x, the more red we use
7  uchar blue = y & 255; // the greater y, the more blue we use
8  unsigned color = rgb(red, 0, blue);
9
10 img[y * DIM + x] = color;
11 }
```

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)

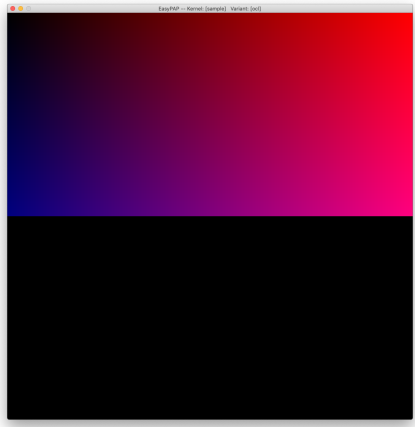
EasyPAP – 2D Kernel – Invocation

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

```
1  unsigned sample_compute_ocl(unsigned nb_iter) {
2    size_t global[2] = {GPU_SIZE_X, GPU_SIZE_Y/2}; // global domain: DIM * (DIM/2) threads
3    size_t local [2] = {TILE_W, TILE_H}; // local domain: groups of TILE_W * TILE_H
4    for (unsigned it = 1; it <= nb_iter; it++) {
5      // set kernel arguments
6      clSetKernelArg(compute_kernel, 0, sizeof (cl_mem), &cur_buffer);
7      // submit the kernel execution command to the queue
8      clEnqueueNDRangeKernel(queue, compute_kernel, 2, NULL, global, local, 0, NULL, NULL);
9    }
10   clFinish (queue); // wait until all commands finished in the queue
11   return 0;
12 }
```

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

```
1  __kernel void sample_ocl(__global unsigned *img) {
2    size_t x = get_global_id(0);
3    size_t y = get_global_id(1);
4
5    uchar red = x & 255; // the greater x, the more red we use
6    uchar blue = y & 255; // the greater y, the more blue we use
7    unsigned color = rgb(red, 0, blue);
8    img[y * DIM + x] = color; // first pixel (x,y)
9
10   // now address the lower half of image
11   size_t y2 = y + get_global_size(1); // y2 = y + 128 in our example
12   uchar blue2 = y2 & 255; color = rgb(red, 0, blue2);
13   img[y2 * DIM + x] = color; // second pixel (x,y2)
14 }
```

Table of Contents

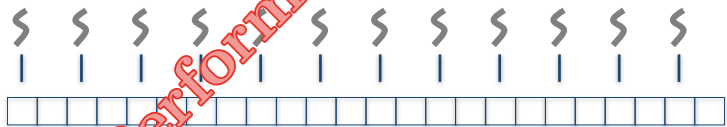
4 Threads and Global Memory Access

Memory Concerns – Scal Vec – Version 2

4 Threads and Global Memory Access

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

```
1 __kernel void scal_vec_v2(__global float *vec, float k)
2 {
3   size_t index = get_global_id(0); // thread id
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 }
```



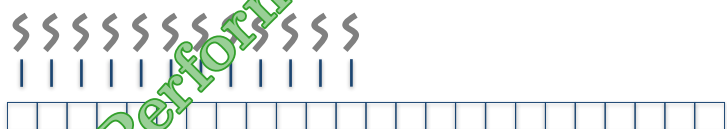
Performance is weak!!

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 work items

```
1 __kernel void scal_vec_v3(__global float *vec, float a)
2 {
3   size_t index = get_global_id(0); // thread id
4   size_t size = get_global_size(0); // #threads
5
6   vec[index       ] = vec[index       ] * a; // 1 load, 1 mult, 1 store
7   vec[index + size] = vec[index + size] * a; // 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 + get_global_id(0)]` is OK
    - ➔ Contiguous threads access contiguous data
  - `vec[2 * get_global_id(0)]` is NOT OK
    - ➔ Contiguous threads access scattered data

Memory Access Coalescing – Sample Example

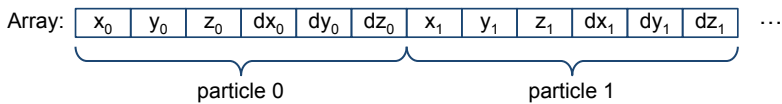
- What if we switch x and y in our `sample` kernel?
  - The output is still correct, but...
  - Performance becomes very weak!
  - Half-warps are contiguous along the x axis
  - But they access vertical columns of data

```
1 __kernel void sample_ocl (__global unsigned *img)
2 {
3   size_t x = get_global_id(0);
4   size_t y = get_global_id(1);
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

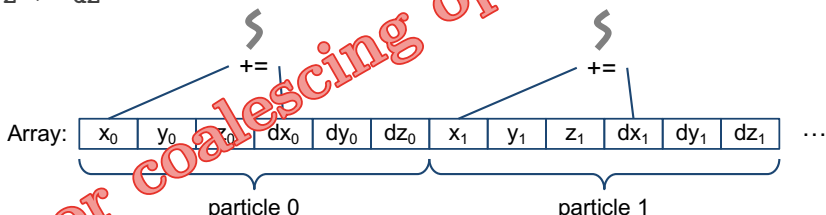
- Example: Moving N particles in a 3D domain
  - Each particle has a position (x, y, z) and a speed vector (d_x, d_y, d_z)
  - We typically use an Array of Structures (aka AoS)
  - Good for cache, isn't it?

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



Memory Access – Data Layout – Part 2

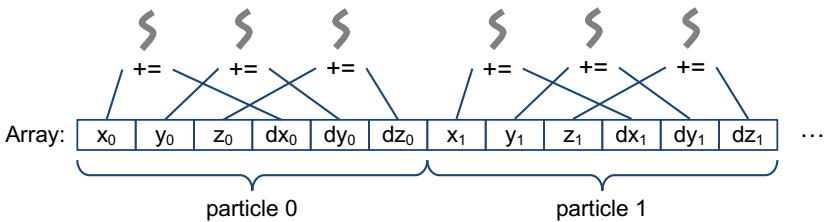
- Moving particles on a GPU
  - One thread per particle
  - `x += dx`
  - `y += dy`
  - `z += dz`



Poor coalescing opportunities!

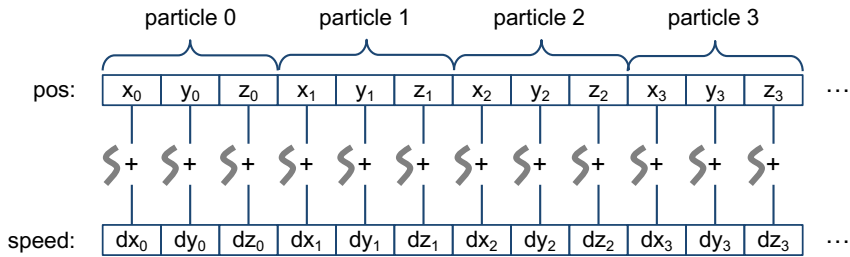
Memory Access – Data Layout – Part 3

- 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...

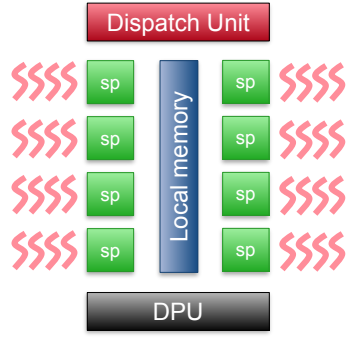
```
1 __kernel void transpose_ocl(__global const unsigned *in, __global unsigned *out)
2 {
3   size_t x = get_global_id(0);
4   size_t y = get_global_id(1);
5
6   out[x * DIM + y] = in[y * DIM + x];
7 }
```

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(...) ...
```

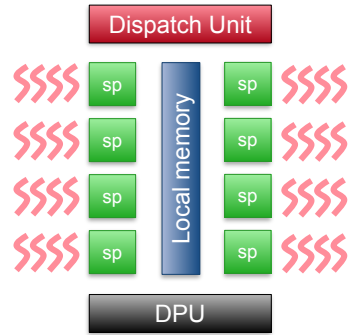


GM80 SM

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 }
```

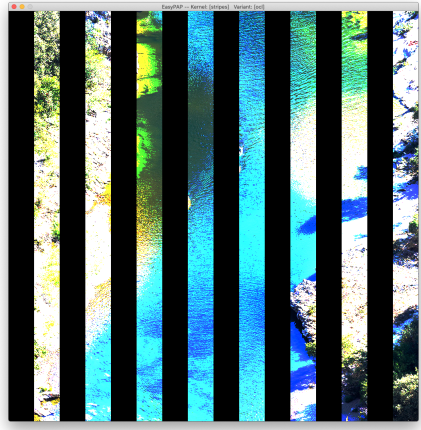
$(x)_{10}$	$(x)_2$	`x & (1 << 0)`
0	00000	00000
1	00001	00001
2	00010	00000
3	00011	00001
4	00100	00000
5	00101	00001
6	00110	00000
7	00111	00001
8	01000	00000
9	01001	00001
10	01010	00000
11	01011	00001
12	01100	00000
13	01101	00001
14	01110	00000
15	01111	00001

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...

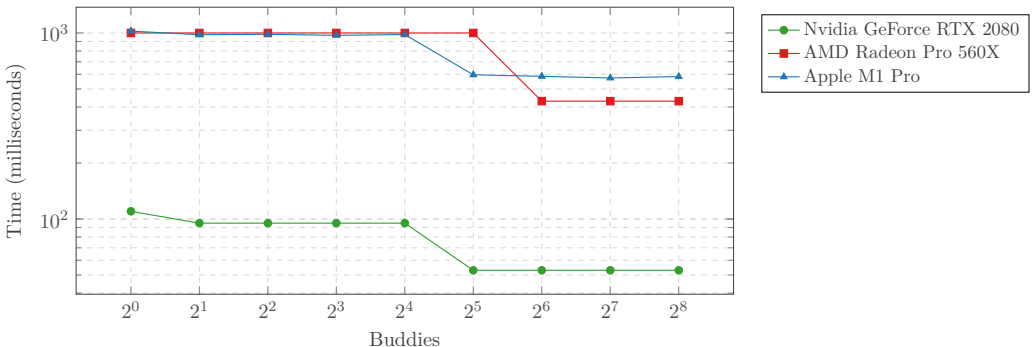
    `./run -l 1024.png -k stripes -g -c 6`



Stripes Kernel – Experiments

5 Threads Divergence

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

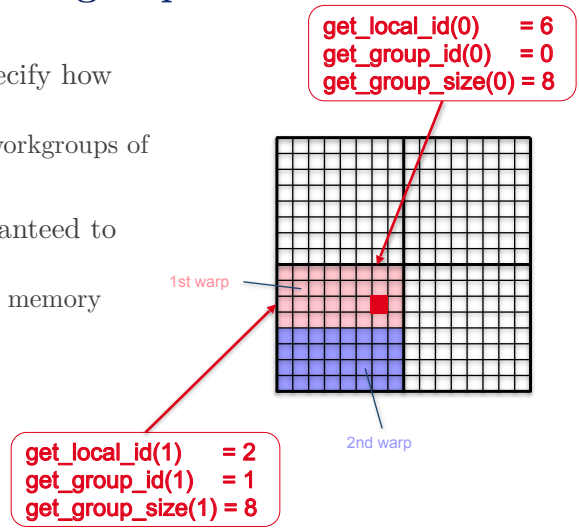


Time for the `stripes` kernel on Nvidia, AMD and M1 Pro GPUs

Table of Contents

Introduction to Workgroups

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

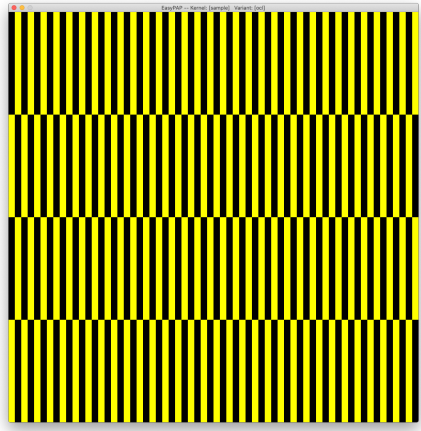


Playing with EasyPAP – Code

6 OpenCL Workgroups

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

```
1  __kernel void sample_ocl(__global unsigned *img)
2  {
3    size_t x = get_global_id(0);
4    size_t y = get_global_id(1);
5
6    if ((get_group_id(0) + get_group_id(1)) % 2)
7      img[y * DIM + x] = rgb(255, 255, 0);
8  }
```



Playing with EasyPAP – Experiments

6 OpenCL Workgroups

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

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

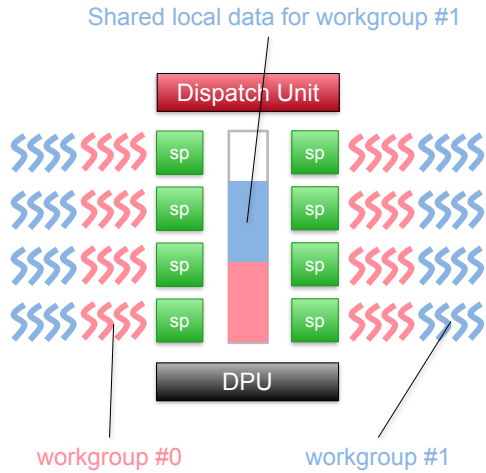
Too much uncoalesced memory accesses!

Table of Contents

7 Local Memory

Introduction to Local Memory

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



Pixelize Kernel – Code

7 Local Memory

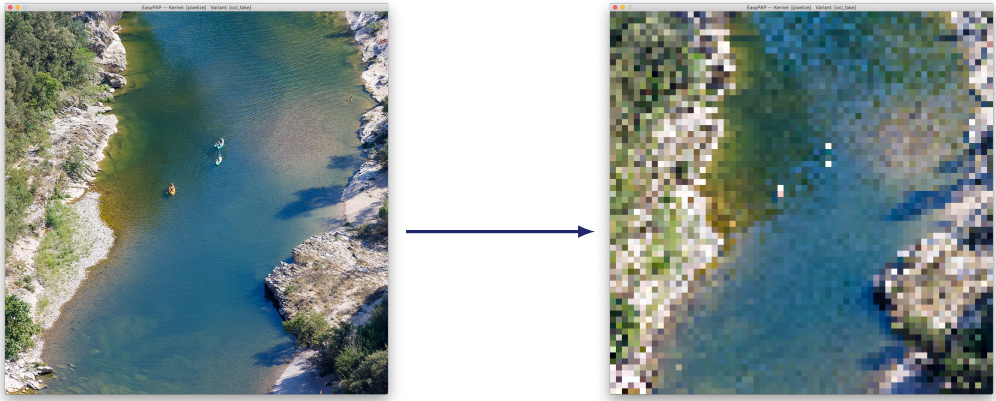
- Local memory is declared inside kernels using `__local`
- Example with this "oversimplified" `pixelize` kernel
  - Each workgroup has its private `color` variable

```
1  __kernel void pixelize_ocl(__global unsigned *img) {
2    size_t xg = get_global_id(0), yg = get_global_id(1);
3    size_t xl = get_local_id (0), yl = get_local_id (1);
4    // declare local memory (shared by all the threads of a workgroup)
5    __local unsigned color;
6    // thread from the upper left corner sets this shared variable
7    if (xl == 0 && yl == 0)
8      color = img[yg * DIM + xg]; // only one thread per wgrp reads from global memory
9    // workgroup threads synchronize to make sure 'color' has been written
10   barrier(CLK_LOCAL_MEM_FENCE);
11   // Finally, all threads set their pixel to this color
12   img[yg * DIM + xg] = color;
13 }
```

Pixelize Kernel – Results

7 Local Memory

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



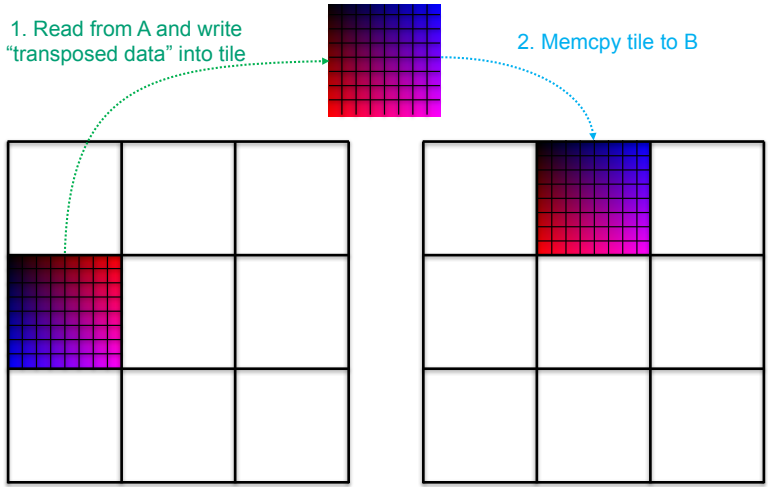
From Pixelize to Transpose Kernel

7 Local Memory

- Workgroups can share more than a scalar value
  - E.g. `__local unsigned tile[TILE_H][TILE_W];`
  - Serves as a “cache” in which data is fetched from global memory
  - 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 local 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

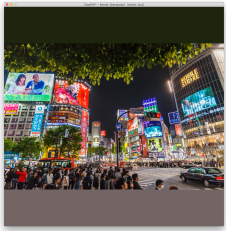
Transpose Kernel – Principle

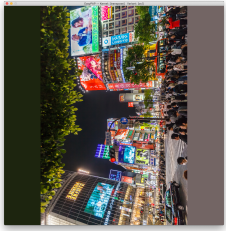
7 Local Memory



Transpose Kernel – Code

7 Local Memory



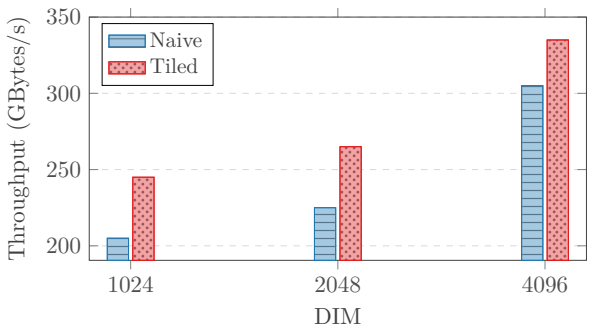


```
./run -k transpose -v ocl_tiled -l shibuya.png -r 1 -g -p
```

```
1  __kernel void transpose_ocl_tiled(
2    __global const unsigned *in,
3    __global       unsigned *out)
4  {
5    __local unsigned tile[TILE_H][TILE_W];
6
7    size_t xg = get_global_id(0), yg = get_global_id(1);
8    size_t xl = get_local_id (0), yl = get_local_id (1);
9
10   tile[xl][yl] = in[yg * DIM + xg];
11
12   barrier(CLK_LOCAL_MEM_FENCE);
13
14   out[(xg - xl + yl) * DIM + yg - yl + xl] = tile[yl][xl];
15 }
```

Transpose Kernel – Experiments 1

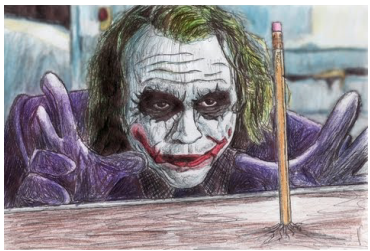

```
./run -g -k transpose -v <ocl_naif|ocl_tiled> -i 1000 -n -s <DIM>
```



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

Transpose Kernel – Magic Trick

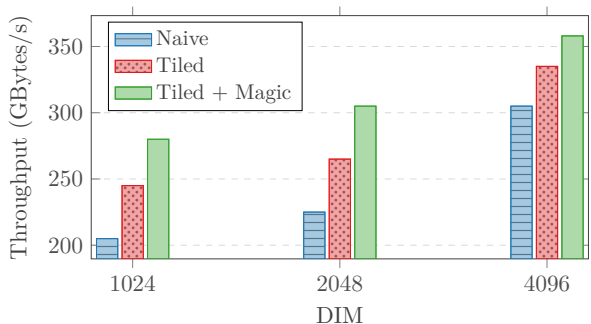
Why the hell do we add this extra column we don't even use?!



```
1  __kernel void transpose_ocl_tiled(
2    __global const unsigned *in,
3    __global       unsigned *out)
4  {
5    __local unsigned tile[TILE_H][TILE_W+1];
6
7    size_t xg = get_global_id(0), yg = get_global_id(1);
8    size_t xl = get_local_id (0), yl = get_local_id (1);
9
10   tile[xl][yl] = in[yg * DIM + xg];
11
12   barrier(CLK_LOCAL_MEM_FENCE);
13
14   out[(xg - xl + yl) * DIM + yg - yl + xl] = tile[yl][xl];
15 }
```

Transpose Kernel – Experiments 2


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

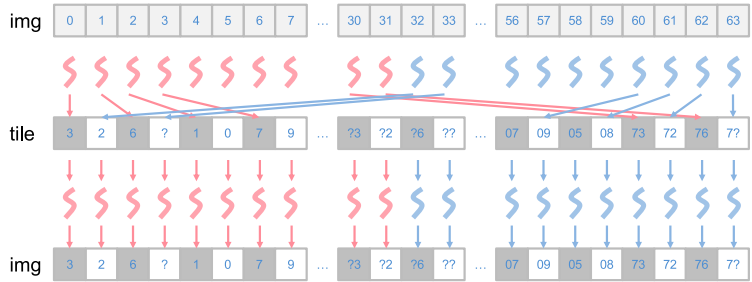


Throughput of the `transpose` kernel on Nvidia GeForce GTX 2080

Stripes Kernel – Principle

7 Local Memory

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



Stripes Kernel – Code

7 Local Memory

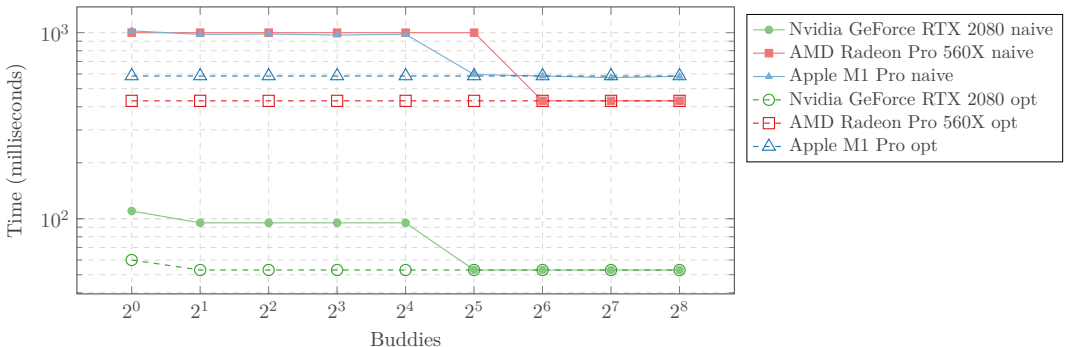
- Implementation of 1-pixel-width stripes
  - Divergence avoiding
  - Memory coalescing
- Do not forget about the synchros!

```
1  __local unsigned tile[TILE_H][TILE_W];
2  size_t xg = get_global_id(0), yg = get_global_id(1),
3  size_t xl = get_local_id (0), yl = get_local_id (1);
4  size_t index = 2 * xl;
5
6  tile[yl][xl] = in[yg * DIM + xg];
7  barrier(CLK_LOCAL_MEM_FENCE);
8
9  if (index < get_local_size(0)) {
10   tile[yl][index] = darken(tile[yl][index]);
11 } else {
12   index += -get_local_size(0) + 1;
13   tile[yl][index] = brighten(tile[yl][index]);
14 }
15
16 barrier(CLK_LOCAL_MEM_FENCE);
17 out[yg * DIM + xg] = tile[yl][xl];
```

Stripes Kernel – Experiments

7 Local Memory

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



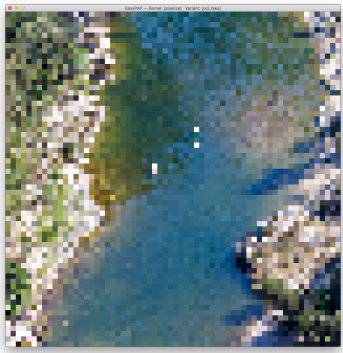
Time for the `stripes` kernel on Nvidia, AMD and M1 Pro GPUs

Table of Contents

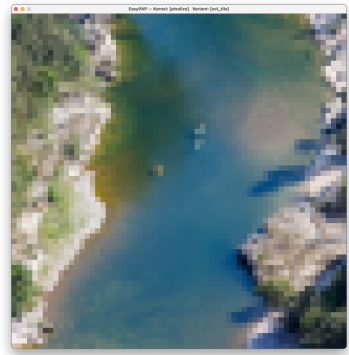
Accurate Pixelize Kernel – Principle

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
  - OpenCL provide “vectors” of 2, 3 or 4 scalar values

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

```
1  __kernel void pixelize_ocl(__global unsigned *in)
2  {
3    __local int4 tile[TILE_H][TILE_W];
4
5    size_t xg = get_global_id(0), yg = get_global_id(1);
6    size_t xl = get_local_id (0), yl = get_local_id (1);
7
8    tile[yl][xl] = color_to_int4(in[yg * DIM + xg]);
9    // ...
10 }
```

Accurate Pixelize Kernel – Reduction

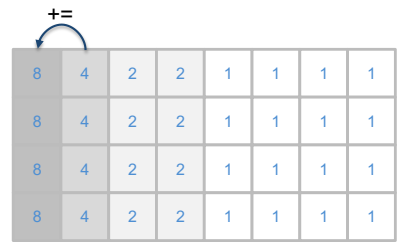
- Reduction
  - We first cache pixels into local memory
  - Then we can perform our 2D reduction inside tile
  - There is one thread per cell
    - To maximize throughput of load operation
- How do we compute the sum of all cells?

```
1  __kernel void pixelize_ocl(__global unsigned *in)
2  {
3    __local int4 tile[TILE_H][TILE_W];
4
5    size_t xg = get_global_id(0), yg = get_global_id(1);
6    size_t xl = get_local_id (0), yl = get_local_id (1);
7
8    tile[yl][xl] = color_to_int4(in[yg * DIM + xg]);
9    barrier (CLK_LOCAL_MEM_FENCE);
10   // ...
11 }
```

Accurate Pixelize Kernel – Horizontal Reduc.

8 Reductions

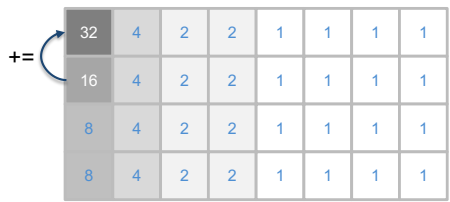
- Let's consider tiles of 8×4 cells
- How do we compute the sum of all cells?
  - Well, we could perform a first wave of 4×4 additions
    - 4 additions per row
    - Half of threads do not work
- Next step
  - Second wave of 2×4 additions
    - 2 additions per row
    - Only 1/4 of threads participate
- 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][xl] += tile[yl][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][0] += tile[yl + 1][0];
4 }
```

Accurate Pixelize Kernel – Code

8 Reductions

```
1  __kernel void pixelize_ocl_accurate(__global unsigned *in) {
2    __local int4 tile[TILE_H][TILE_W];
3    size_t xg = get_global_id(0); size_t yg = get_global_id(1);
4    size_t xl = get_local_id (0); size_t yl = get_local_id (1);
5
6    tile[yl][xl] = color_to_int4(in[yg * DIM + xg]);
7
8    // Averaging each line
9    for (int d = TILE_W >> 1; d > 0; d >>= 1) {
10     barrier(CLK_LOCAL_MEM_FENCE);
11     if (xl < d)
12       tile[yl][xl] += tile[yl][xl + d];
13   }
14   // Averaging first column only
15   for (int d = TILE_H >> 1; d > 0; d >>= 1) {
16     barrier(CLK_LOCAL_MEM_FENCE);
17     if (xl == 0 && yl < d)
18       tile[yl][xl] += tile[yl + d][xl];
19   }
20
21   barrier(CLK_LOCAL_MEM_FENCE);
22   in[yg * DIM + xg] = int4_to_color(tile[0][0] / (int4)(TILE_W * TILE_H));
23 }
```

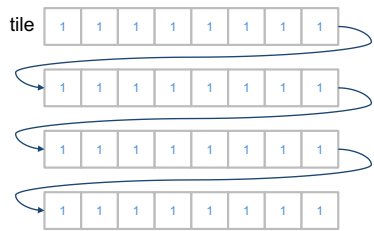
Accurate Pixelize Kernel – Alternative

8 Reductions

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

```
1 __local int4 tile[TILE_H * TILE_W];
2 size_t xl = get_local_id(1) * TILE_W +
3             get_local_id(0);
```

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



From 2D to 1D

Accurate Pixelize Kernel – Code

8 Reductions

```
1  __kernel void pixelize_ocl_accurate(__global unsigned *in)
2  {
3    __local int4 tile[TILE_H * TILE_W];
4    size_t xg = get_global_id(0), yg = get_global_id(1);
5    size_t xl = get_local_id(1) * GPU_TILE_W + get_local_id(0);
6
7    tile[xl] = color_to_int4(in[yg * DIM + xg]);
8
9    for (int d = (TILE_W * TILE_H) >> 1; d > 0; d >>= 1) {
10     barrier(CLK_LOCAL_MEM_FENCE);
11     if (xl < d)
12       tile[xl] += tile[xl + d];
13   }
14   barrier(CLK_LOCAL_MEM_FENCE);
15
16   in[yg * DIM + xg] = int4_to_color(tile[0] / (int4) (TILE_W * TILE_H));
17 }
```

Final Notes about Reductions

- Workgroup-wide reductions are part of OpenCL 2.1 specification
  - Few implementations available :-(
  - Too bad, because it is supported by most hardware
- For reduction on large data sets (> workgroup max size), multi-pass kernels must be used
  - No accelerator-wide barrier
  - Barrier between successive kernels
    - Each kernel performs separate per-workgroup reductions, and write results in memory
    - Loop until #elements $\leq$ workgroup size

Table of Contents

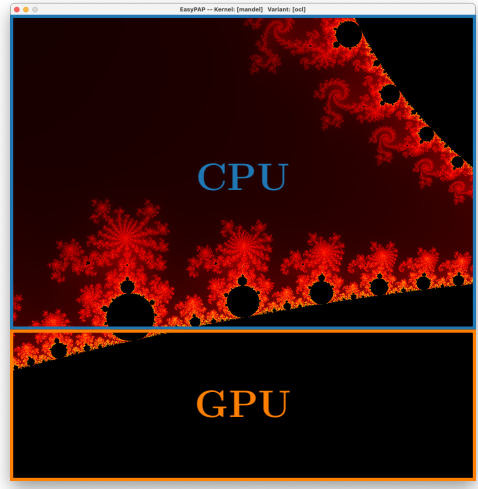
Example with the Mandelbrot Kernel

- 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!

```
1  __kernel void mandel_ocl(__global unsigned *img,
2                           float leftX, float xstep,
3                           float topY, float ystep,
4                           unsigned MAX_ITERATIONS) {
5    size_t i = get_global_id(1), j = get_global_id(0);
6
7    float xc = leftX + xstep * j;
8    float yc = topY - ystep * i;
9    float x = 0.0, y = 0.0; // Z = X + i*Y
10
11   unsigned iter;
12   for (iter = 0; iter < MAX_ITERATIONS; iter++) {
13     float x2 = x * x, y2 = y * y;
14     if (x2 + y2 > 4.0) // Stop iterations when |Z| > 2
15       break;
16
17     float twoxy = (float)2.0 * x * y;
18     x = x2 - y2 + xc;
19     y = twoxy + yc;
20   }
21
22   img[i * DIM + j] = (iter < MAX_ITERATIONS) ?
23     /* then */ mandel_iter2color(iter) :
24     /* else */ 0x000000FF; // black
25 }
```

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?