

Programming GPU Accelerators with OpenCL

**Raymond Namyst
Pierre-André Wacrenier**

**Background:
From early video coprocessors
to current GPUs**

Background

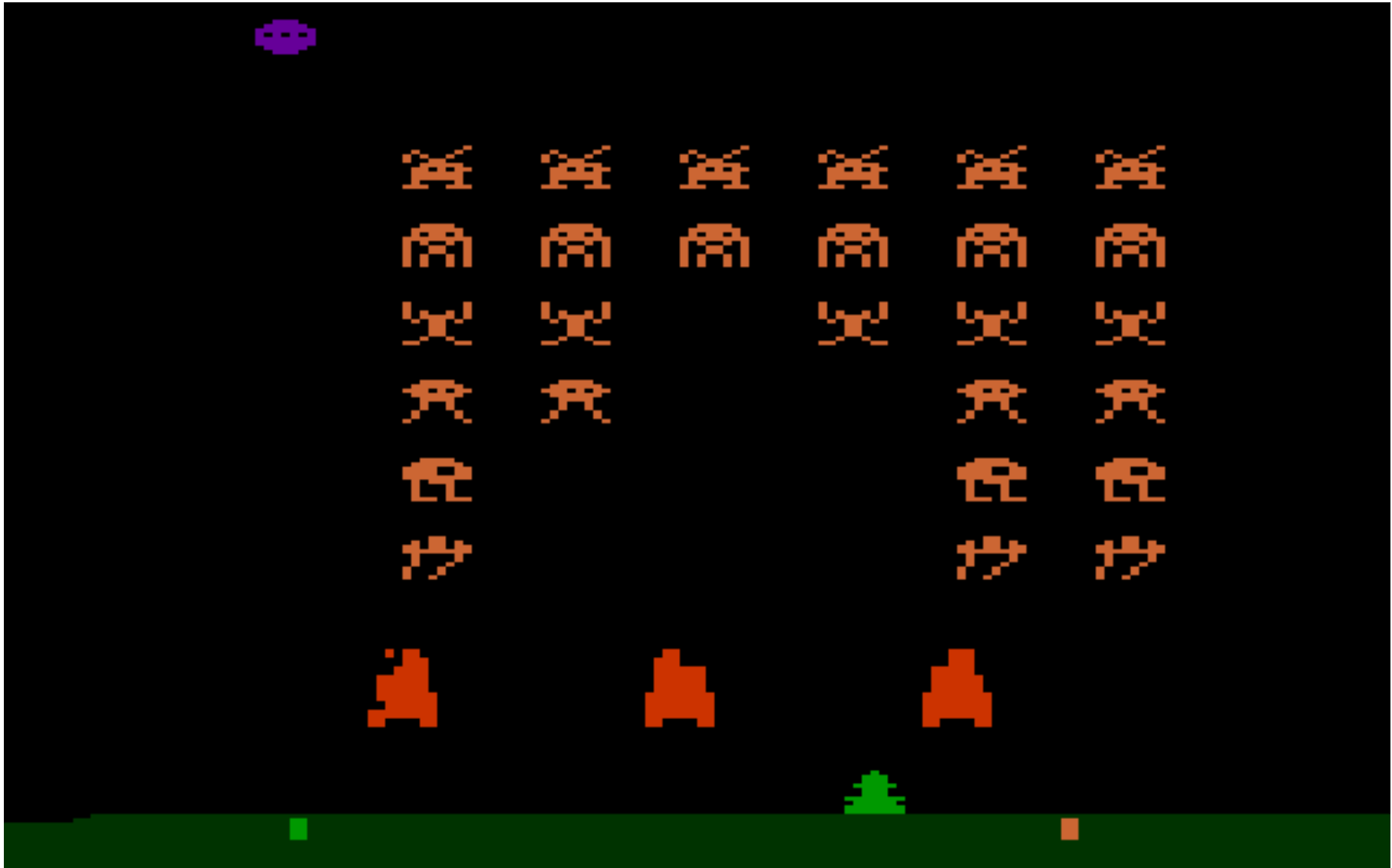
A brief history of GPUs

- Please read the nice exhaustive history at TechSpot:
 - <https://www.techspot.com/article/650-history-of-the-gpu/>
- First Graphic Coprocessors were obviously 2D
 - 1976: RCA “Pixie” video chip (CDP1861), 64x128 pixels
 - 1977: Television Interface Adapter (TIA) 1A
 - Integrated into *Atari 2600*
 - RAMless: no framebuffer!
 - 320x240 pixels
 - 1981: Motorola MC6845 (IBM PC, Apple II)
 - Character-based display



A Touch of Nostalgia

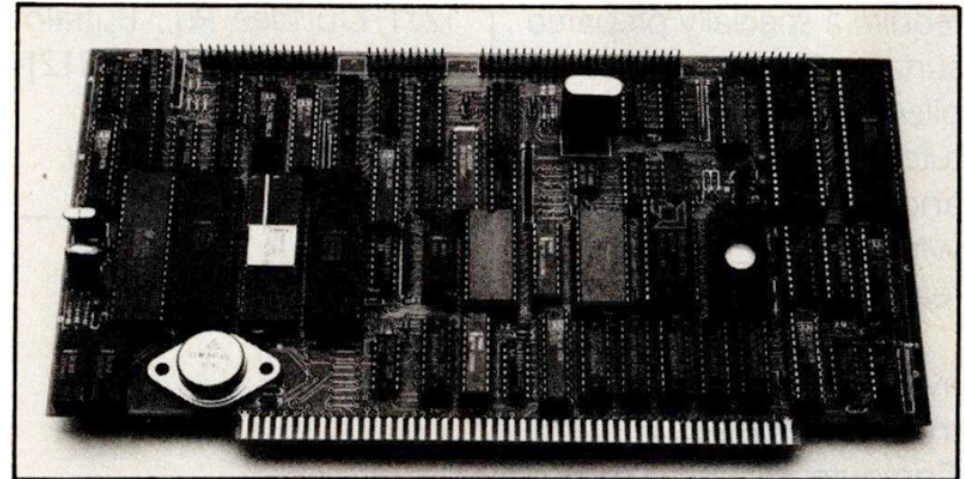
Space Invaders on Atari 2600



Background

A brief history of GPUs

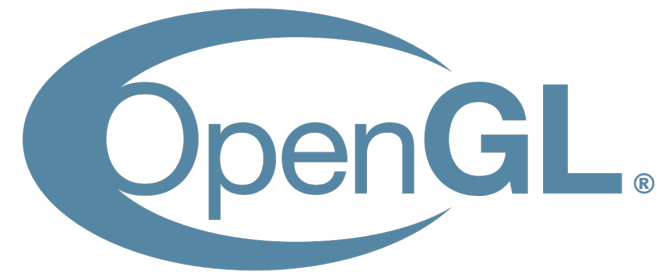
- 1983: *Intel iSBX 275 Video Graphics Controller Multimode Board*
 - 256x256, 8 colors
 - 512x512 monochromic
 - Lines, rectangles, circles...
 - Hardware zooming & scrolling
 - 32KB, \$1000 😊
 - Professional flight simulators
- 1987: *ATI EGA Wonder*
 - 640x350, 16 colors
 - 256KB of DRAM
 - \$399
 - EGA Wonder 800
 - 800x600 VGA
 - \$449



Background

- 1992, OpenGL 1.0 released by SGI
 - multi-platform API for both 2D and 3D graphics
 - Initially aimed at Unix
 - Quickly adopted for 3D gaming
- 1992: Wolfenstein 3D (Id Software)
 - First “First-Person Shooter”
<https://classicreload.com/wolfenstein-3d.html>
- 1993: Birth of *Nvidia*
- 1995: *Microsoft* promotes its Direct3D API
 - But also supports OpenGL

A brief history of GPUs



Background

A brief history of GPUs

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



SEGA Virtua Fighter Remix for Diamond Edge3D (NV1)

Background

- 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)
 - 533MB/s (against 132MB/s using PCI)
 - NB: later, cards will embed fast GDDR memory
 - Nvidia Riva 128
 - Quake 2, Quake 3...
- 1998
 - 3dfx Voodoo 2
 - 800x600
 - New landmark in framerates for many games
 - Scan Line Interleave (SLI)
Aggregate multiple cards via a ribbon cable
 - Intel i740 (worth mentioning 😊)

A brief history of GPUs



Background

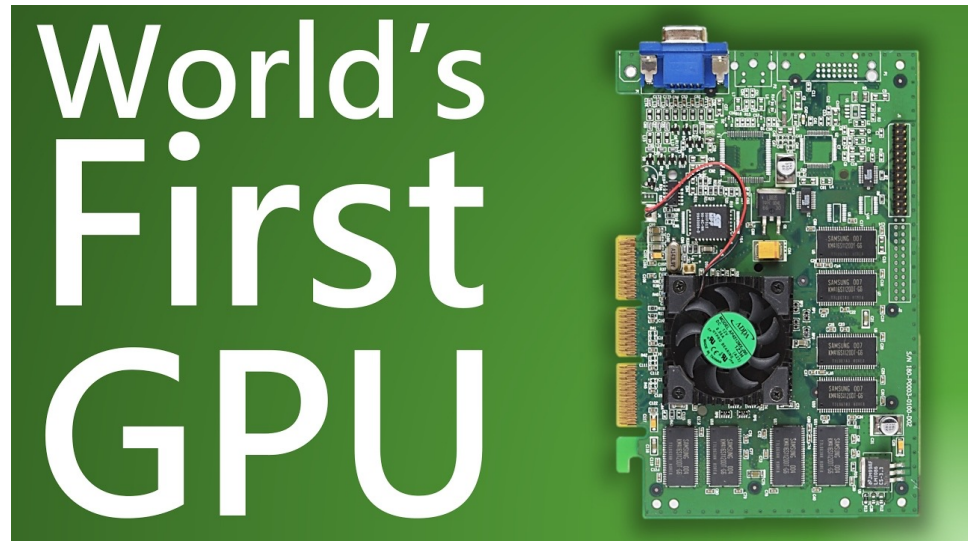
A brief history of GPUs



Background

A brief history of GPUs

- 1998
 - Sega chooses PowerVR (instead of 3dfx) for its Dreamcast console...
 - Microsoft Direct3D gains popularity
 - 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



Background

GPU Accelerators

- 2001
 - Nvidia GeForce 3 (NV20)
 - Programmable units
“*shaders*”
- 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



TECHPOWERUP

GPU Computing about to become mainstream?

GPU Accelerators

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



GPU need specific programming environments

Think “highly parallel”!

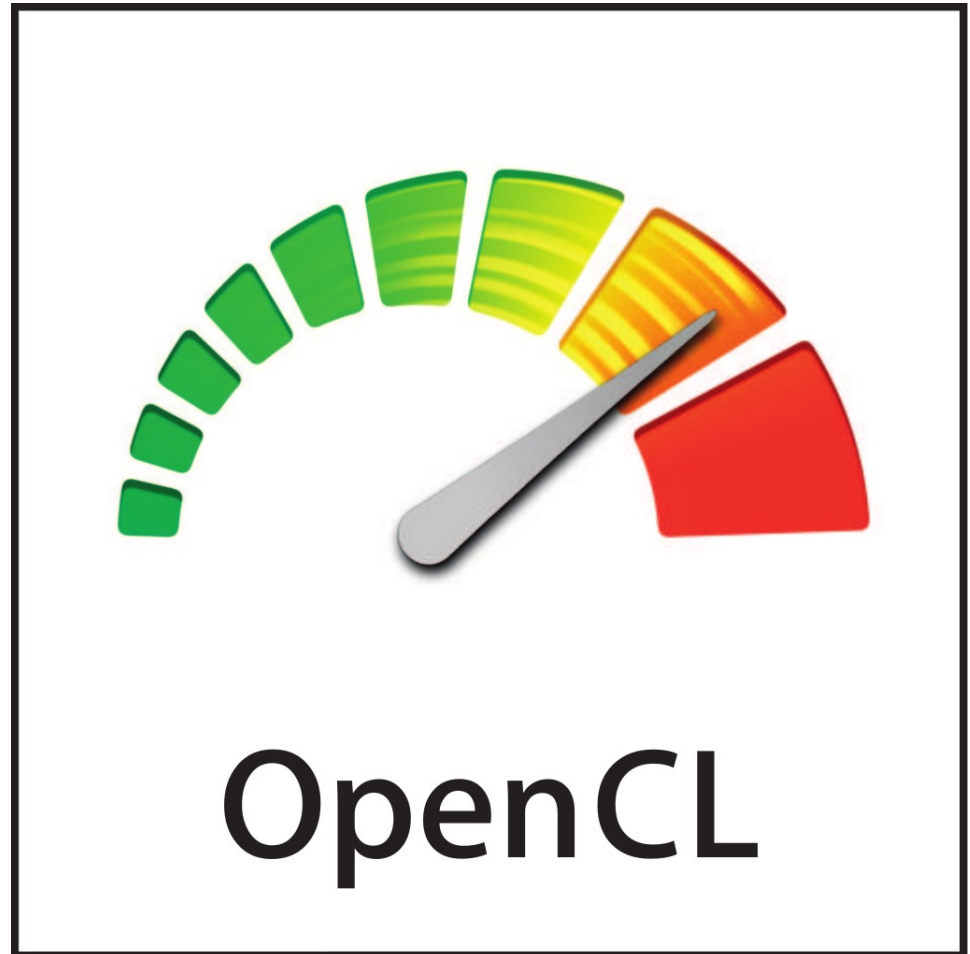
- GPU feature many processors
 - 5000+ in Nvidia Tesla V100 (Volta)
 - At each cycle, many processors execute the same instruction on different data
 - “Simple Instruction – Multiple Data” execution model (SIMD)
 - GPUs require massive parallelism to achieve high performance
- GPU have on-board memory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



GPU Computing about to become mainstream?

GPU Accelerators

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



The OpenCL Programming Environment

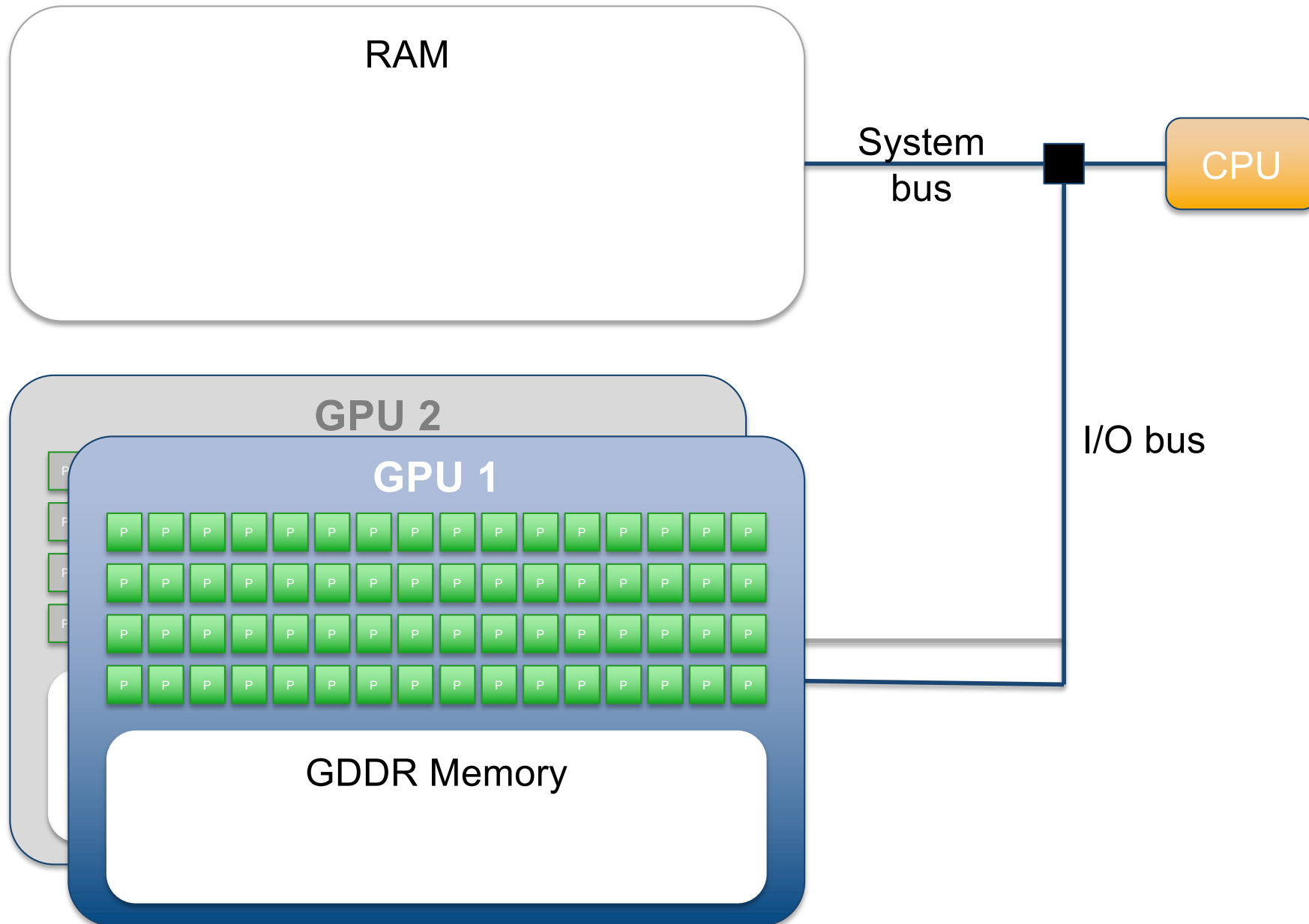
OpenCL

Overview

- 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 \approx main function of a C program
 - Can be invoked from CPU side
 - Notable differences:
 - Kernels are executed in parallel by many threads

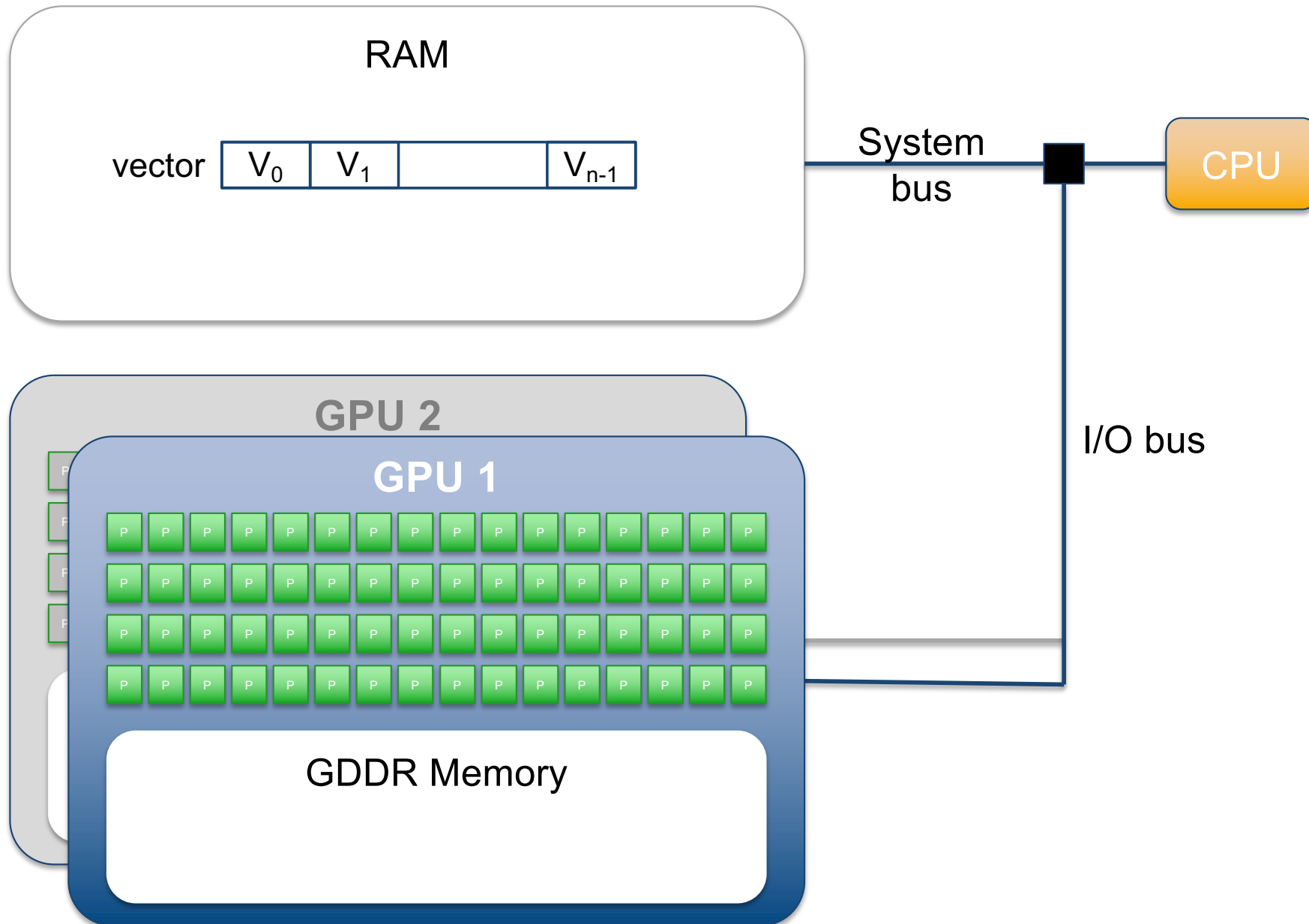
Programming with OpenCL

The big picture



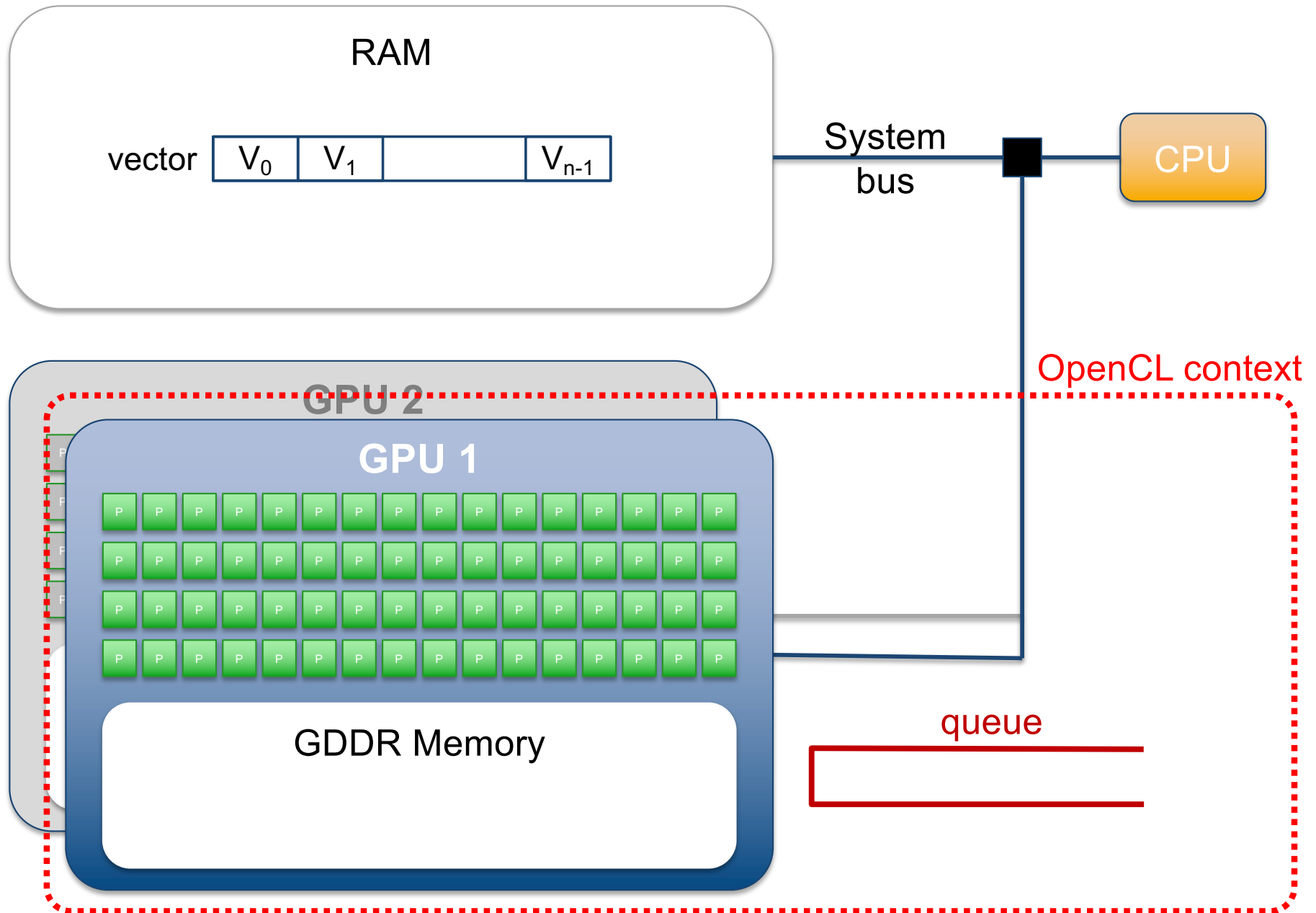
Programming with OpenCL

How to modify our vector on GPU1?



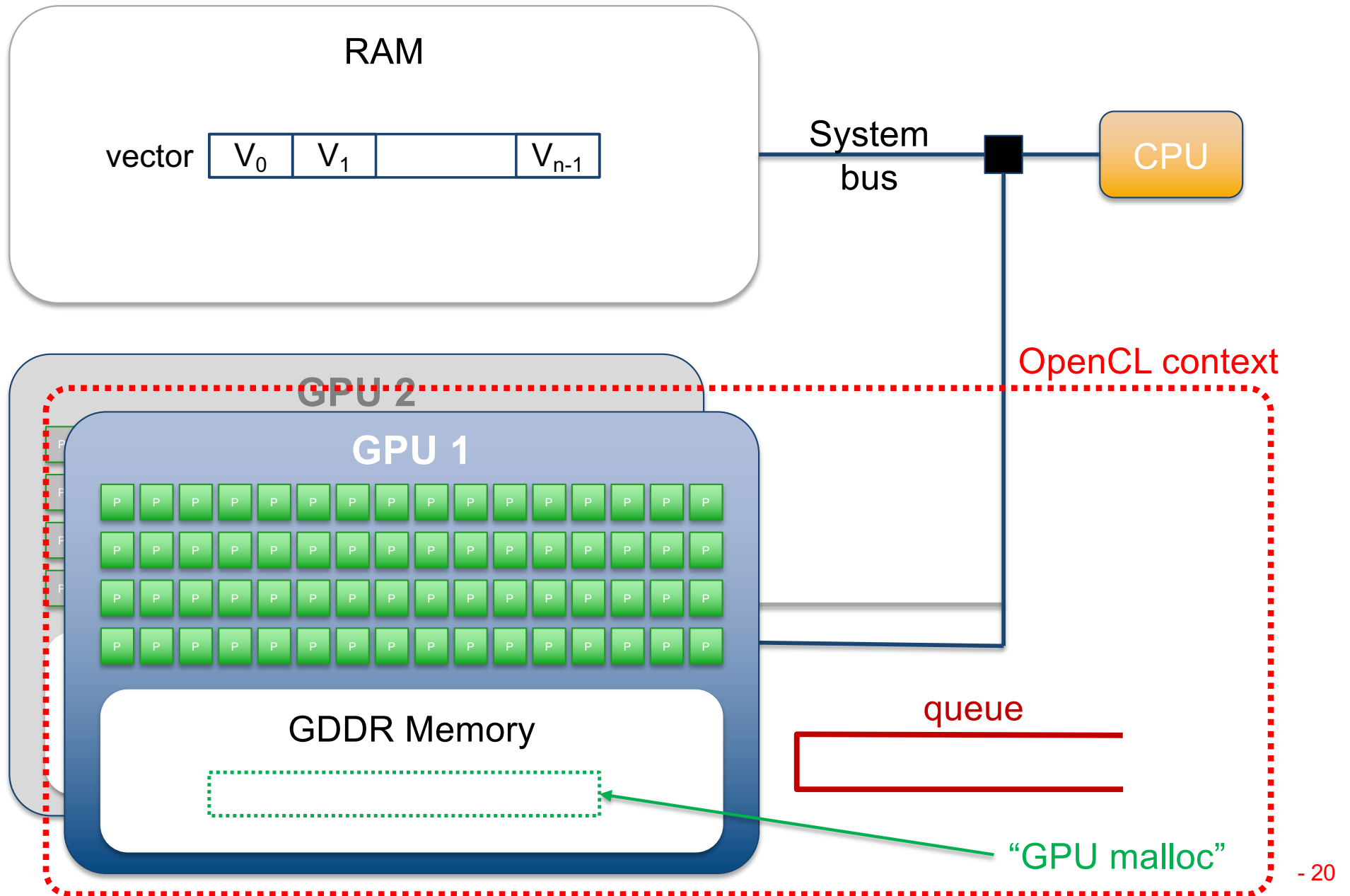
Programming with OpenCL

1) Setup OpenCL context and work queue



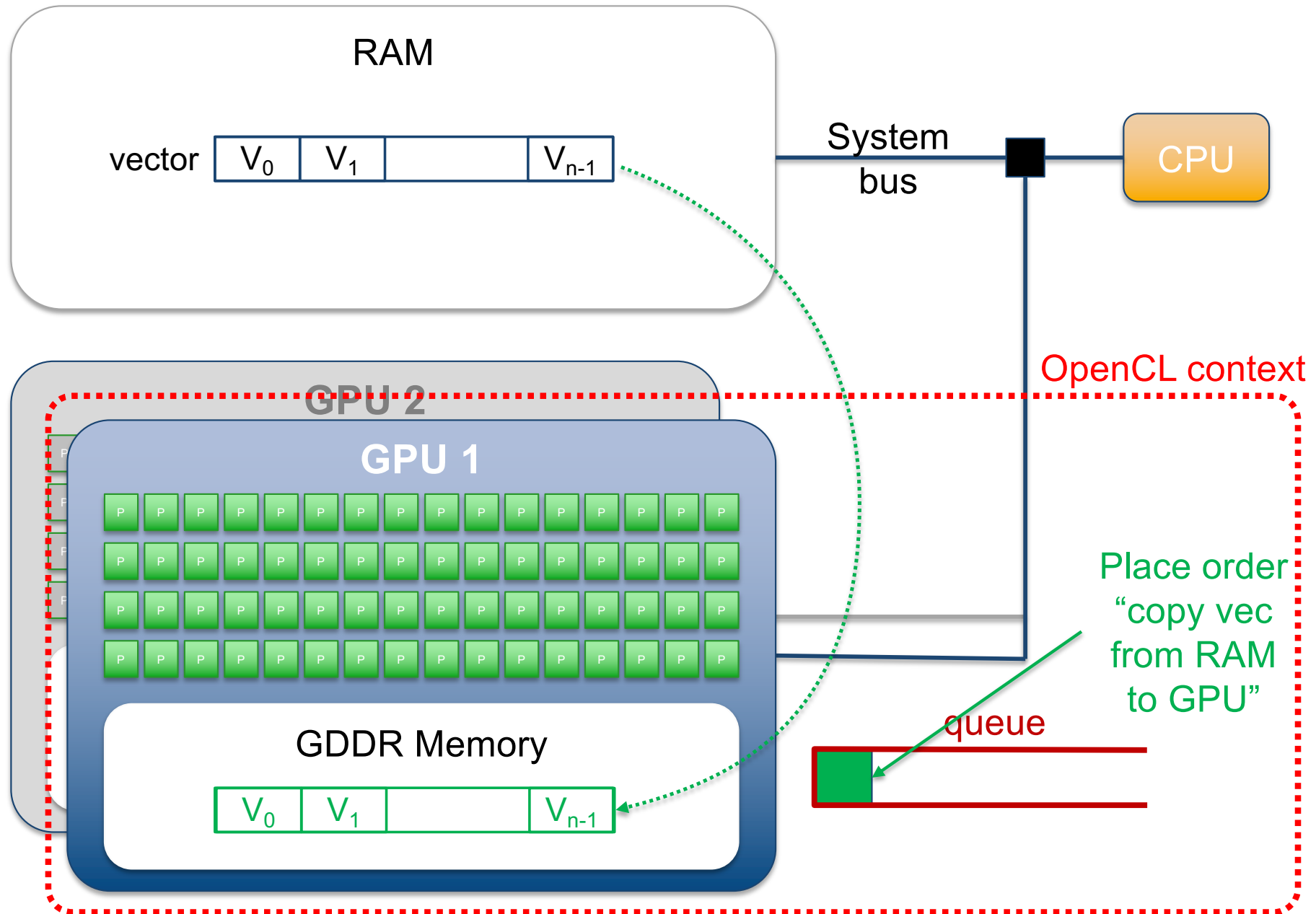
Programming with OpenCL

2) Allocate memory on GPU



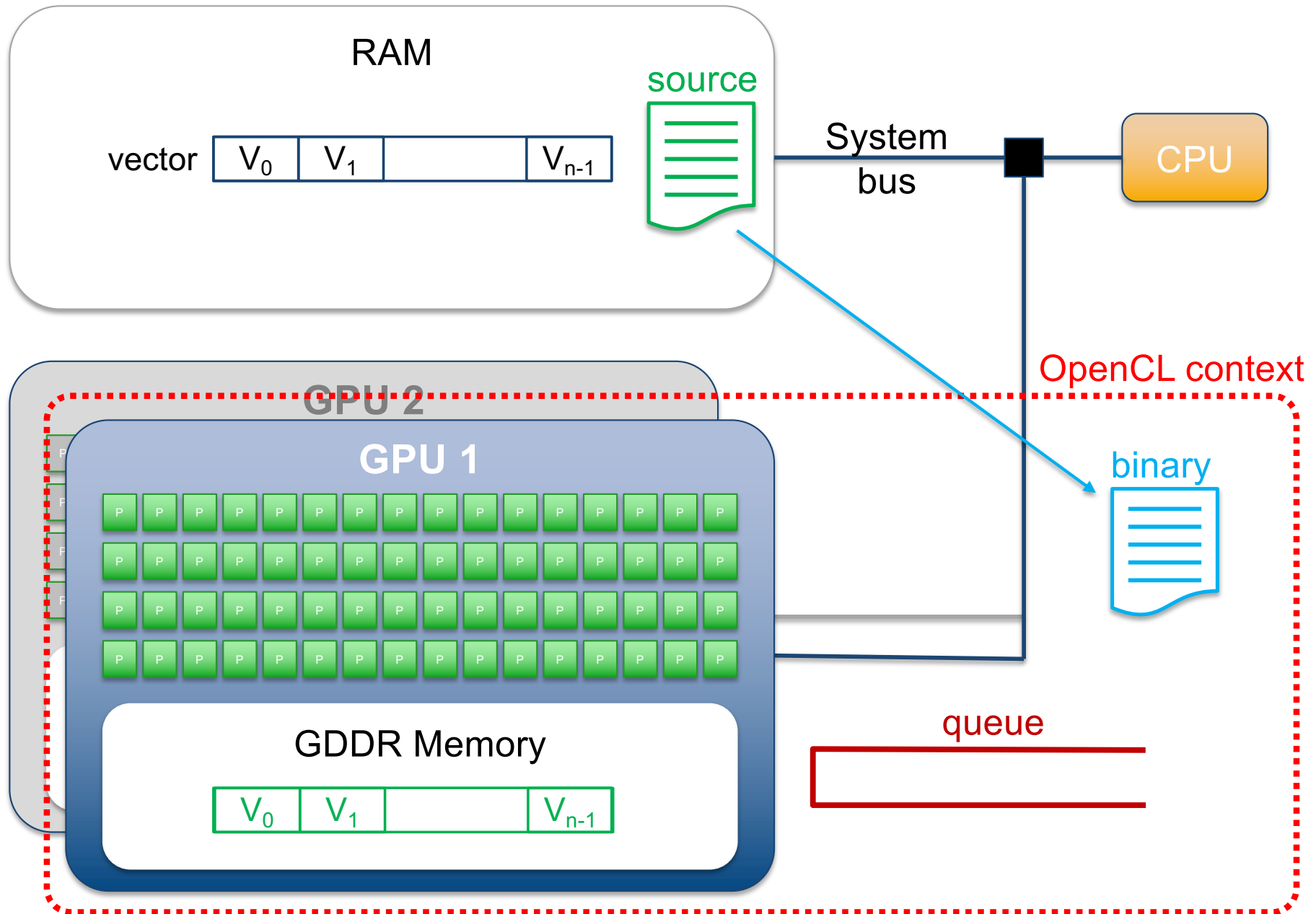
Programming with OpenCL

3) Send data to GPU



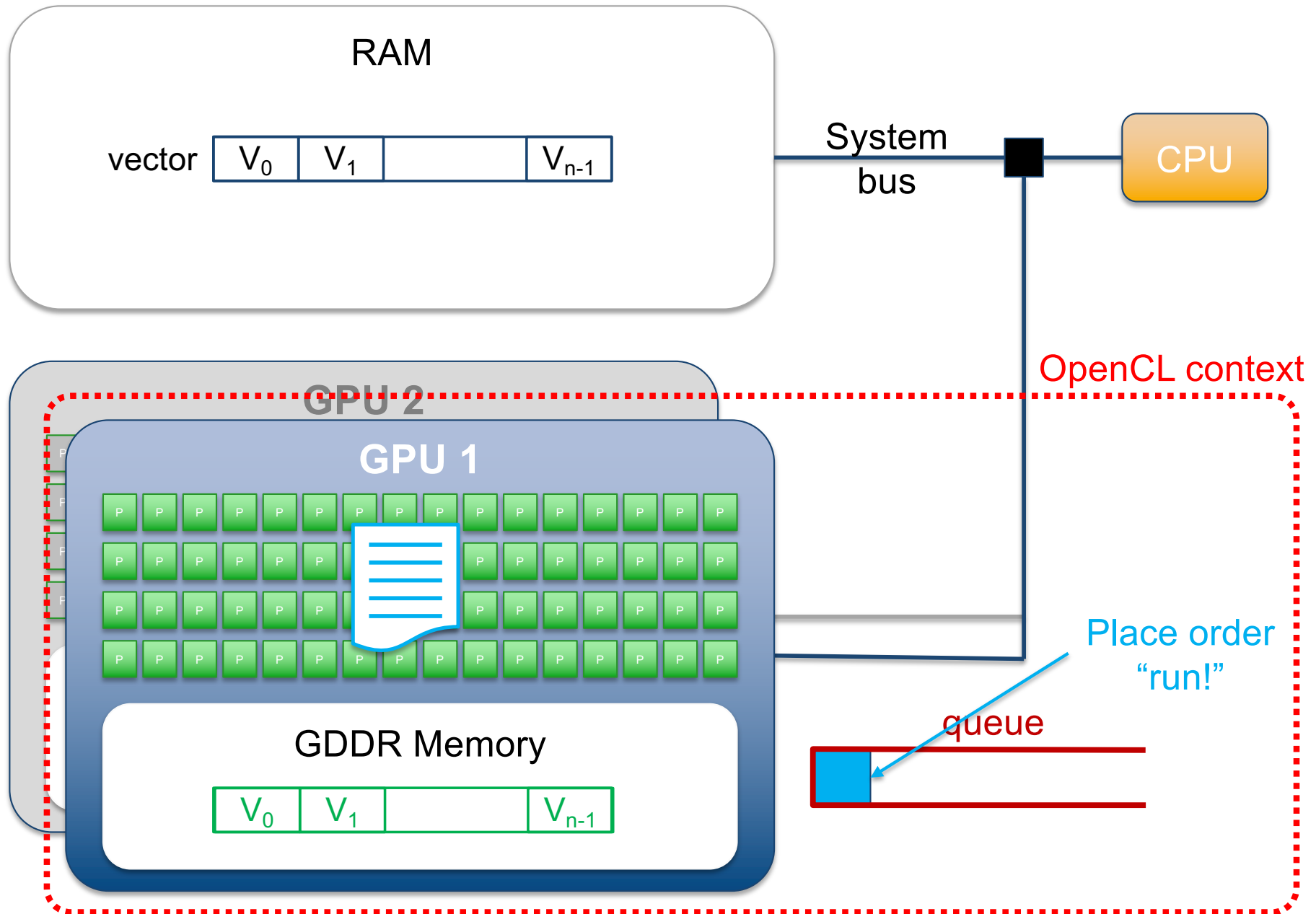
Programming with OpenCL

4) Compile OpenCL "kernel"



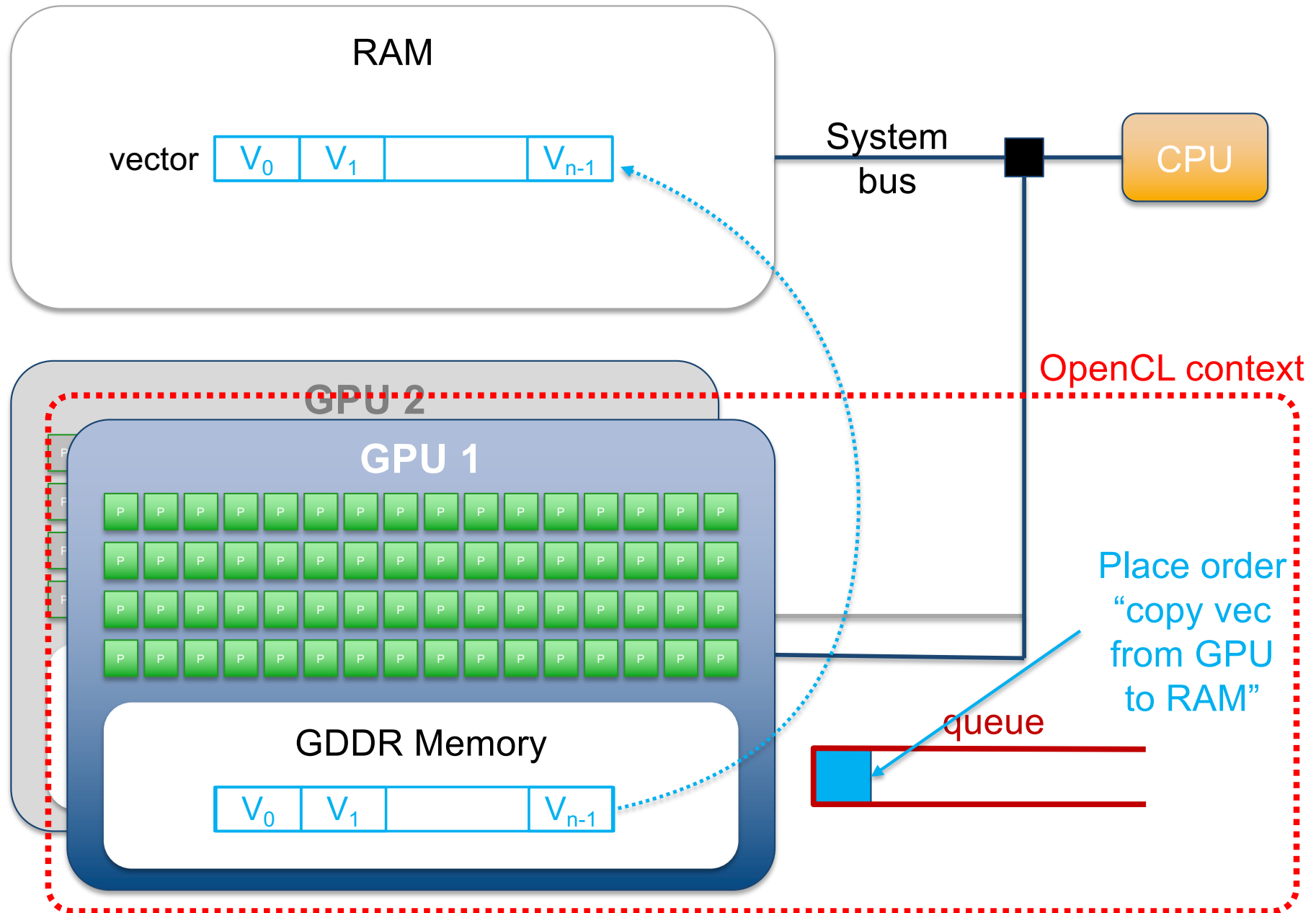
Programming with OpenCL

5) Execute kernel on GPU



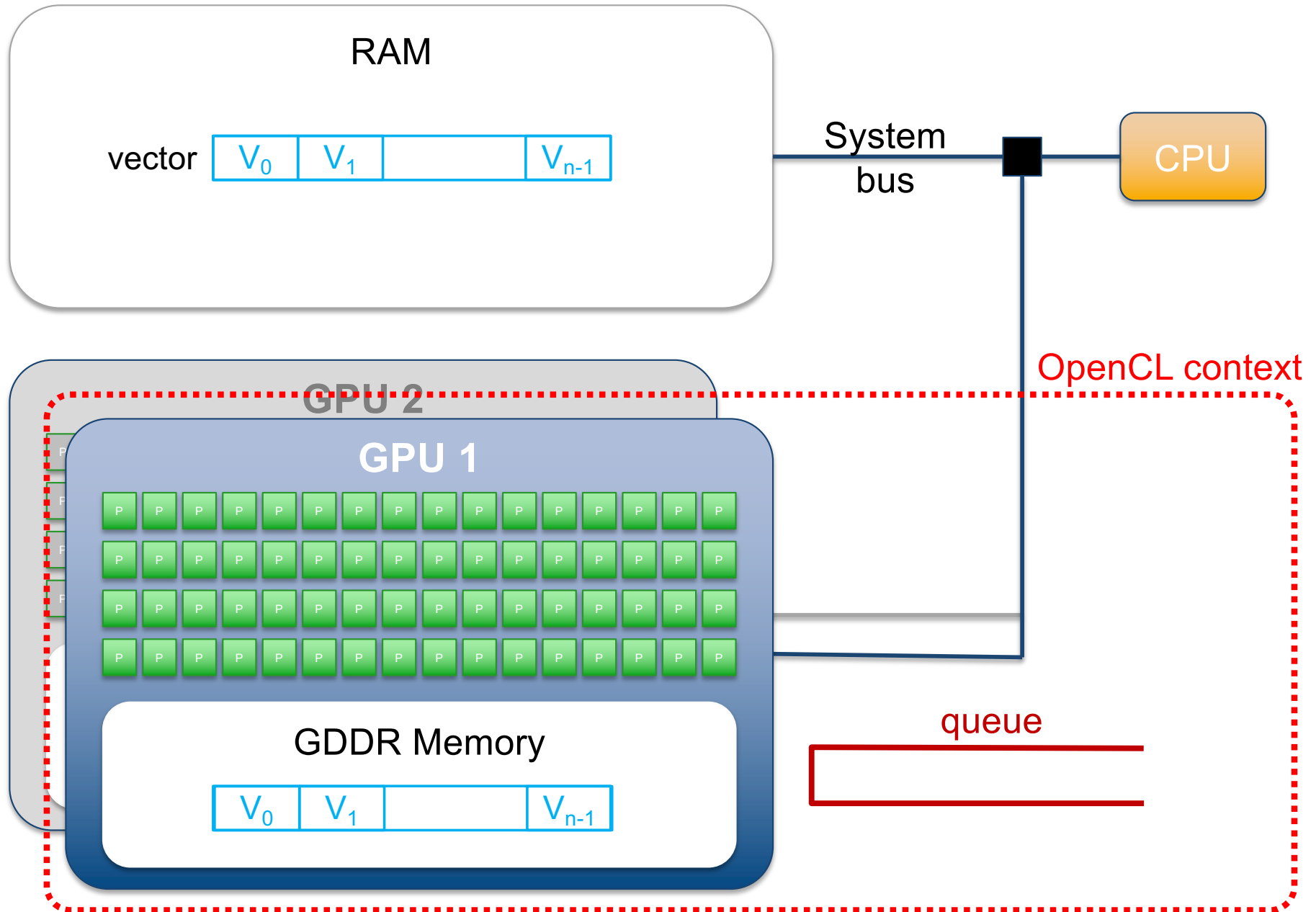
Programming with OpenCL

6) Retrieve data back to RAM



Programming with OpenCL

7) Enjoy your vector! 😊



Programming with OpenCL

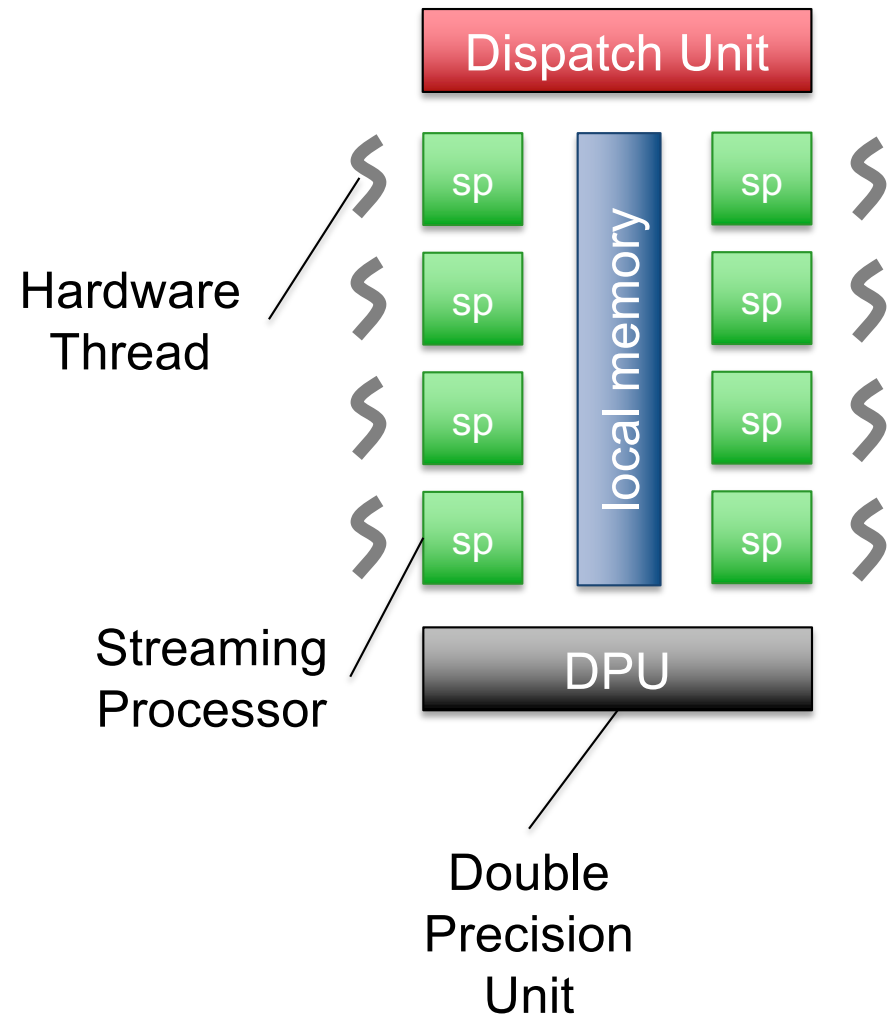
Typical workflow of a simple OpenCL program

- 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!
- Before we explore the OpenCL programming language, we need to understand the execution model of GPUs

GPU execution model

Illustration with a good old Nvidia GPU

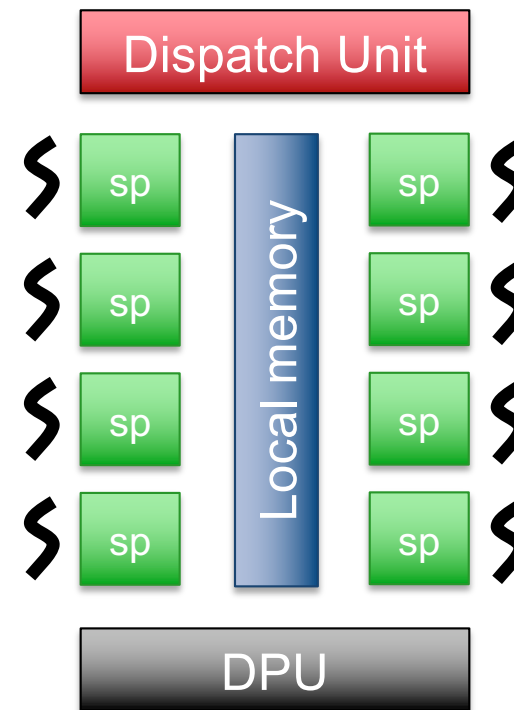
- Basic block = Streaming Multiprocessor (see Figure)
 - SM are clusters of 8 Streaming Processors
 - Local memory sharing
 - Synchronization
- Streaming Processor
 - 64 KB registers!
 - Threads are just “sets of registers”
 - Creation/destruction is free!
 - Interleaved execution of sequential hardware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



GPU execution model

Illustration with a good old Nvidia GPU

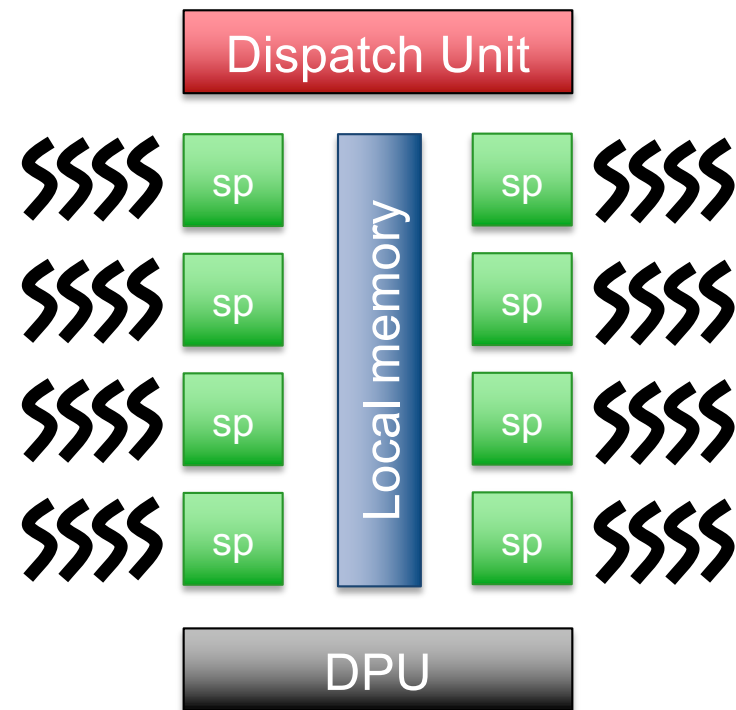
- Only one instruction dispatch unit per Streaming Multiprocessor
 - All SP execute the same instruction at the same clock cycle
 - On different data
= Simple Instruction Multiple Data (SIMD)
- The Dispatch Unit takes 4 cycles to fetch & decode instructions



GPU execution model

Illustration with a good old Nvidia GPU

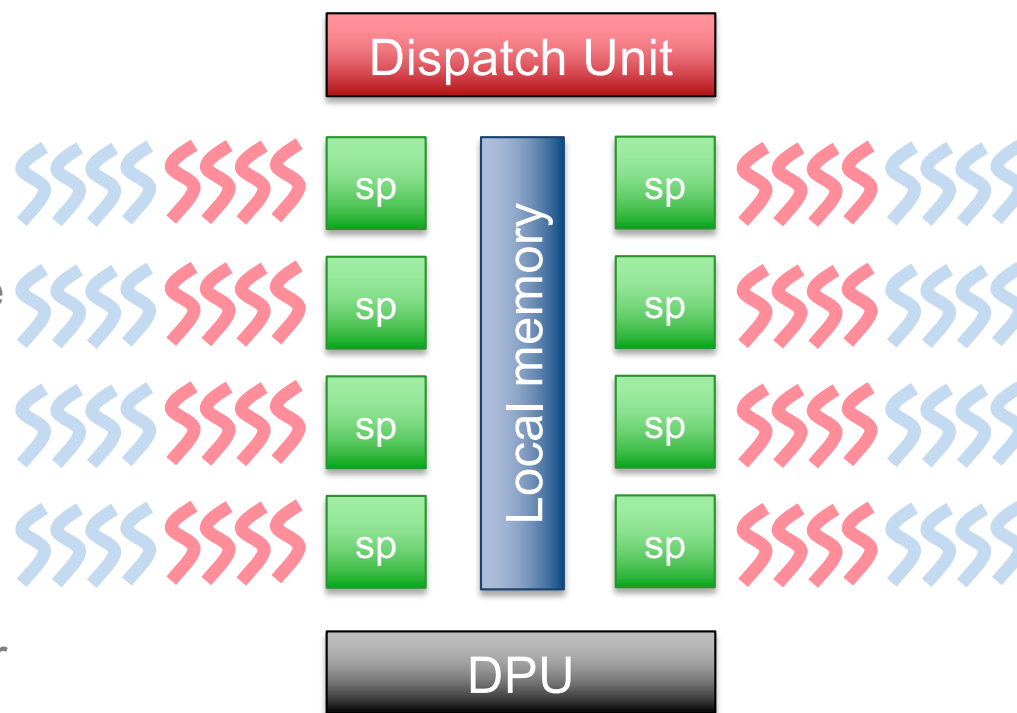
- Only one instruction dispatch unit per Streaming Multiprocessor
 - All SP execute the same instruction at the same clock cycle
 - On different data
= Simple Instruction Multiple Data (SIMD)
- The Dispatch Unit takes 4 cycles to fetch & decode instructions
 - 4 sets of 8 threads are scheduled in a row,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



NVIDIA GPU Execution Model

Warps and half-warps

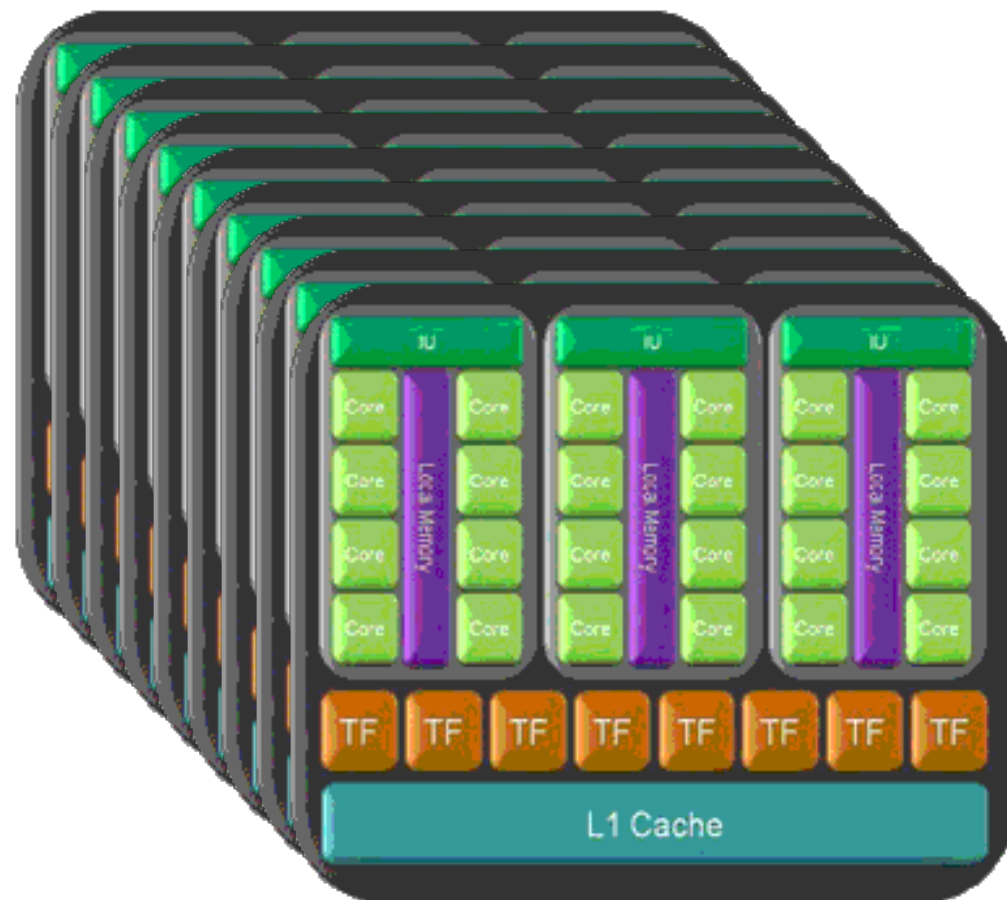
- Threads are implicitly grouped in “warps”
 - **Warp = 32 threads (Nvidia)**
 - Note: on AMD cards, it is called a *wavefront* and it groups 64 threads
 - All threads of the same warp execute the same instruction at the same logical cycle
 - **No divergence!**
- Loading data from global memory is expensive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



NVIDIA GPU Execution Model

Illustration with a good old Nvidia GPU

- GPU = set of Streaming Multiprocessors sharing a global memory
- Nvidia GTX240
 - 30 SM
 - $8 \times 30 = 240$ processors
 - 128 threads max per processor = 30,720 threads!
- Not exactly the usual meaning of “*thread*”...
 - Data-parallelism
 - Regular access patterns



Programming with OpenCL

Let us come back and detail the typical OpenCL workflow

- 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!

Programming with OpenCL

Let us come back and detail the typical OpenCL workflow

- An OpenCL program typically follows these steps:
 1. Configure an OpenCL “queue” (among many other things...)
 - See `ocl_init` function in `src/ocl.c` (EasyPAP)
Yep, it is as tedious as C socket creation 😊
 2. Allocate memory on GPU side
 - See `ocl_alloc_buffers` function in `src/ocl.c` (EasyPAP)
 - `clCreateBuffer` returns an opaque type...
(obviously not usable as a pointer)
...to be later used as a parameter for kernel execution
 3. Transfer (copy) input data from RAM to GPU memory
 - `clEnqueueWriteBuffer` can be synchronous/asynchronous
 - See end of `ocl_send_image` function

Programming with OpenCL

Let us come back and detail the typical OpenCL workflow

- An OpenCL program typically follows these steps (continued) :
 4. Compile kernel for the target GPU architecture
 - See beginning of `ocl_send_image function` (so weird..!)
 - `clCreateProgramWithSource`: attach program source (i.e. a `char *` pointer) to context
 - `clBuildProgram`: compile program for all devices in context
 5. Execute kernel on GPU
 - See next slides 😊
 6. Retrieve output data (copy) from GPU memory to RAM
 - `clEnqueueReadBuffer`
 7. Use the results!

The OpenCL programming language

Kernels

- OpenCL 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

The OpenCL programming language

Kernels

- 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)
 - How threads should be grouped in so-called OpenCL workgroups
 - The role of workgroups will be discussed later

The OpenCL programming language

Thread numbering

- Example: kernel working on a 24x24 matrix, with one thread per cell (576 threads)

- Domain dimensions

2

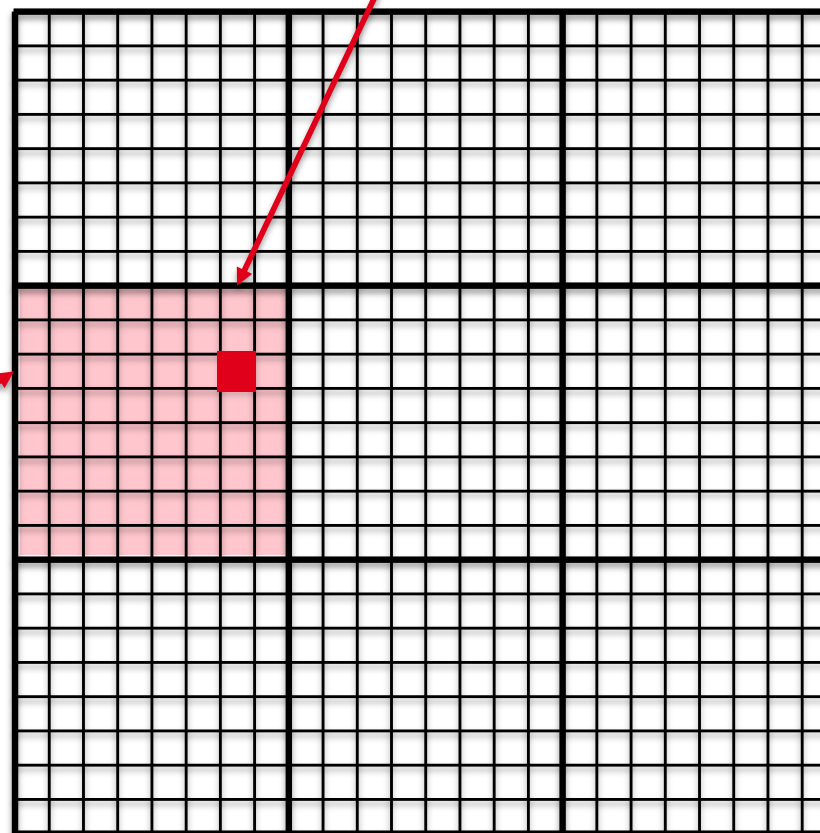
- #threads along each dim

24

- #group_size along each dim

8

```
get_global_id(0) = 6  
get_local_id(0)  = 6  
get_group_id(0)  = 0
```



```
get_global_id(1) = 10  
get_local_id(1)  = 2  
get_group_id(1)  = 1
```

The OpenCL programming language

ScalVec: a simple 1D “scalar.vector” kernel

- “ScalVec” 1D kernel
 - Vector “vec” lies in GPU’s global memory (hence “`__global`”)
 - The kernel is executed with one thread per vector element

```
__kernel void ScalVec(__global float *vec, float k)
{
    int index = get_global_id(0); // thread id

    vec[index] *= k;
}
```

- No loop!
 - Each thread handles one vector element... and that’s it!

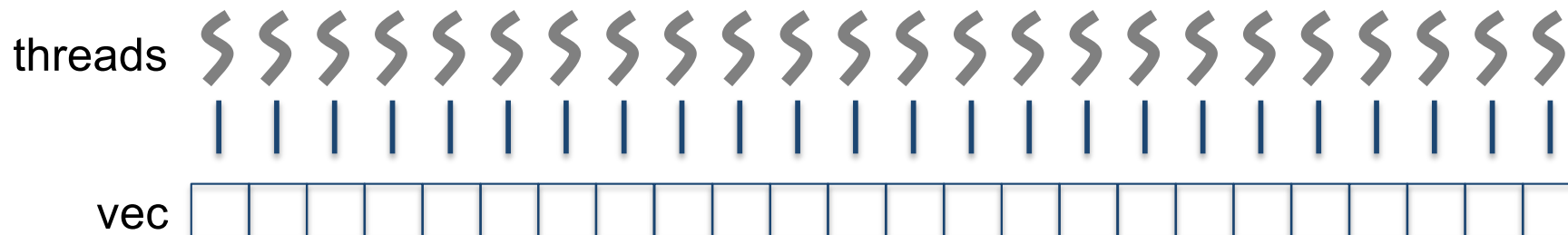
The OpenCL programming language

ScalVec: a simple 1D “scalar.vector” kernel

- “ScalVec” 1D kernel
 - Vector “vec” lies in GPU’s global memory (hence “`__global`”)
 - The kernel is executed with one thread per vector element

```
__kernel void ScalVec(__global float *vec, float k)
{
    int index = get_global_id(0); // thread id

    vec[index] *= k;
}
```



OpenCL with EasyPAP

2D kernels

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

```
unsigned sample_invoke_ocl (unsigned nb_iter)
{
    size_t global[2] = {GPU_SIZE_X, GPU_SIZE_Y}; // global domain size for our calculation
    size_t local[2]  = {TILE_W, TILE_H}; // local domain size for our calculation
    cl_int err;

    for (unsigned it = 1; it <= nb_iter; it++) {
        // Set kernel arguments
        err = 0;
        err |= clSetKernelArg (compute_kernel, 0, sizeof (cl_mem), &cur_buffer);
        check (err, "Failed to set kernel arguments");

        err = clEnqueueNDRangeKernel (queue, compute_kernel, 2, NULL, global, local, 0, NULL, NULL);
        check (err, "Failed to execute kernel");
    }
    return 0;
}
```


OpenCL with EasyPAP

2D kernels

- 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 $y * DIM + x$

```
#include "kernel/ocl/common.cl"

__kernel void sample_ocl (__global unsigned *img)
{
    int x = get_global_id (0);
    int y = get_global_id (1);

    unsigned color = 0xFFFF00FF; // yellow

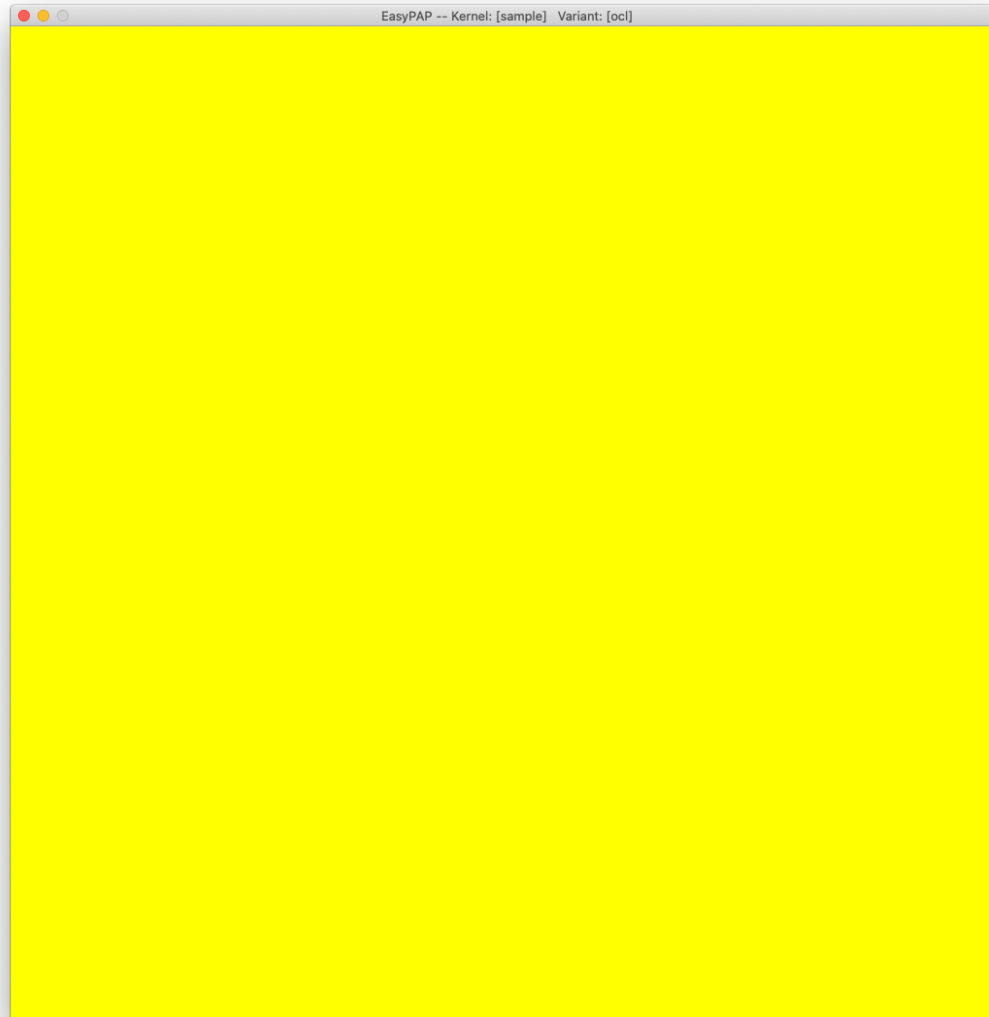
    img [y * DIM + x] = color;
}
```

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

OpenCL with EasyPAP

2D kernels

- Let's see how it works on a 256x256 image:
 - `./run -s 256 -k sample --gpu`



OpenCL with EasyPAP

2D kernels

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

```
unsigned sample_invoke_ocl (unsigned nb_iter)
{
    size_t global[1] = {DIM * DIM}; // DIM * DIM workitems
    size_t local[1]  = {TILE_W * TILE_H};
    cl_int err;

    for (unsigned it = 1; it <= nb_iter; it++) {
        // Set kernel arguments
        err = 0;
        err |= clSetKernelArg (compute_kernel, 0, sizeof (cl_mem), &cur_buffer);
        check (err, "Failed to set kernel arguments");

        err = clEnqueueNDRangeKernel (queue, compute_kernel, 1, NULL, global, local, 0, NULL, NULL);
        check (err, "Failed to execute kernel");
    }
    return 0;
}
```

OpenCL with EasyPAP

2D kernels

- 1D version of kernel/ocl/sample.cl:

```
#include "kernel/ocl/common.cl"

__kernel void sample_ocl (__global unsigned *img)
{
    int index = get_global_id (0);

    unsigned color = 0xFFFF00FF; // yellow

    img [index] = color;
}
```

Output is identical
(trust me 😊)

OpenCL with EasyPAP

2D kernels

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

```
#include "kernel/ocl/common.cl"

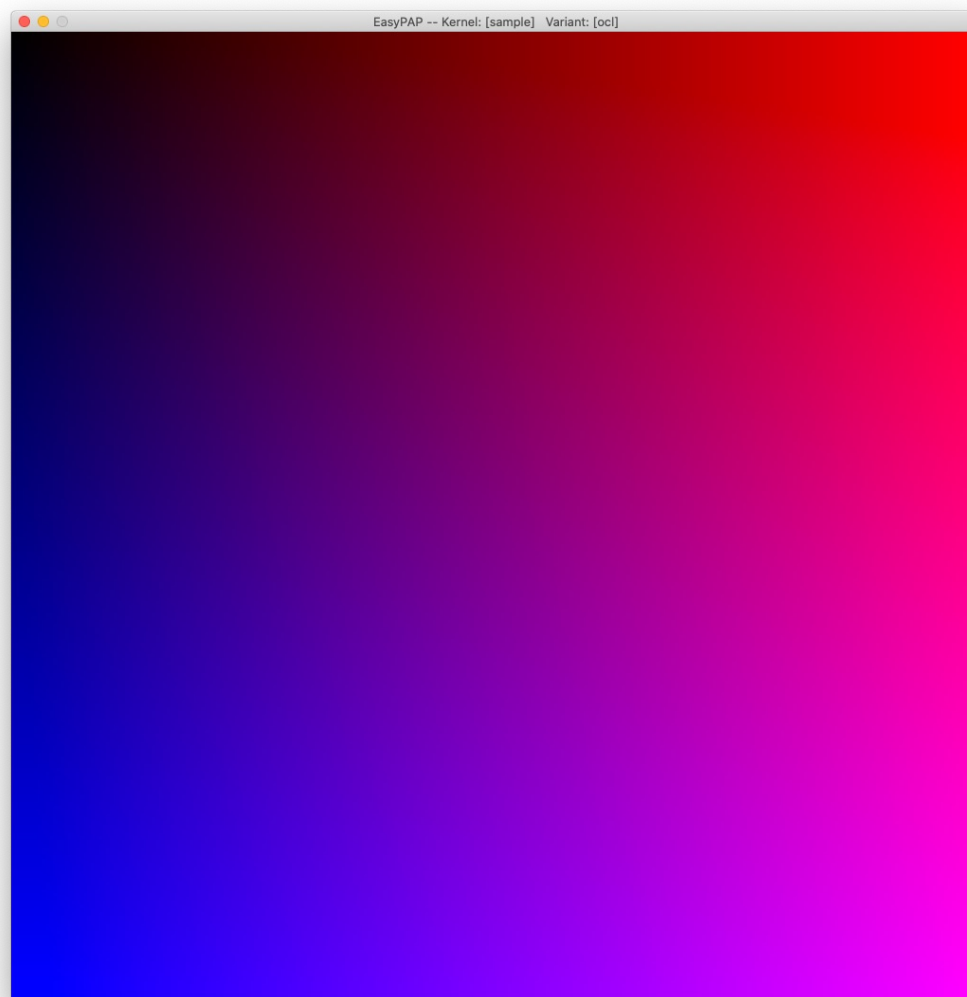
__kernel void sample_ocl (__global unsigned *img)
{
    int x = get_global_id (0);
    int y = get_global_id (1);
    unsigned color = 0xFF; // opacity = 100%
    color |= (x & 255) << 24; // the greater x, the more red we use
    color |= (y & 255) << 8; // the greater y, the more blue we use
    img [y * DIM + x] = color;
}
```

By the way: we use (... & 255) in case the kernel is executed on images larger than 256x256...

OpenCL with EasyPAP

2D kernels

- We run the program the same way (no need to type 'make' 😊)
 - `./run -s 256 -k sample -g`

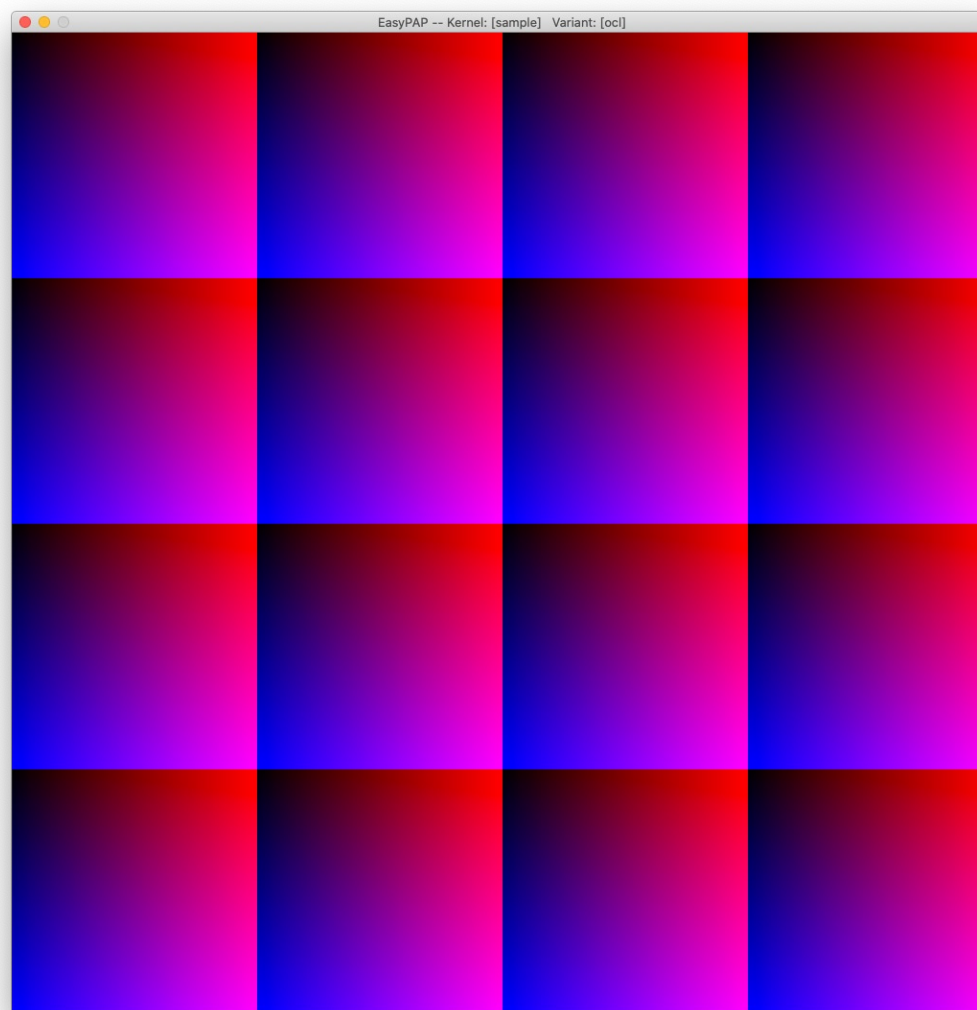


OpenCL with EasyPAP

2D kernels

- Great! Let us see how it works on larger images

```
- ./run -s 1024 -k sample -g
```



OpenCL with EasyPAP

2D kernels

- So far so good... But what if we ask for 4096^2 threads?
 - `./run -s 4096 -k sample -g`



How can that be???

OpenCL with EasyPAP

2D kernels

- 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 x 2048 \approx 160K threads
 - At least, not simultaneously!

OpenCL with EasyPAP

2D kernels

- 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 x 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)

OpenCL with EasyPAP

2D kernels

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

```
unsigned sample_invoke_ocl (unsigned nb_iter)
{
    size_t global[2] = {GPU_SIZE_X, GPU_SIZE_Y / 2}; // global domain : DIM * DIM / 2 threads
    size_t local[2]  = {TILE_W, TILE_H}; // local domain : groups of TILEX * TILEY
    cl_int err;

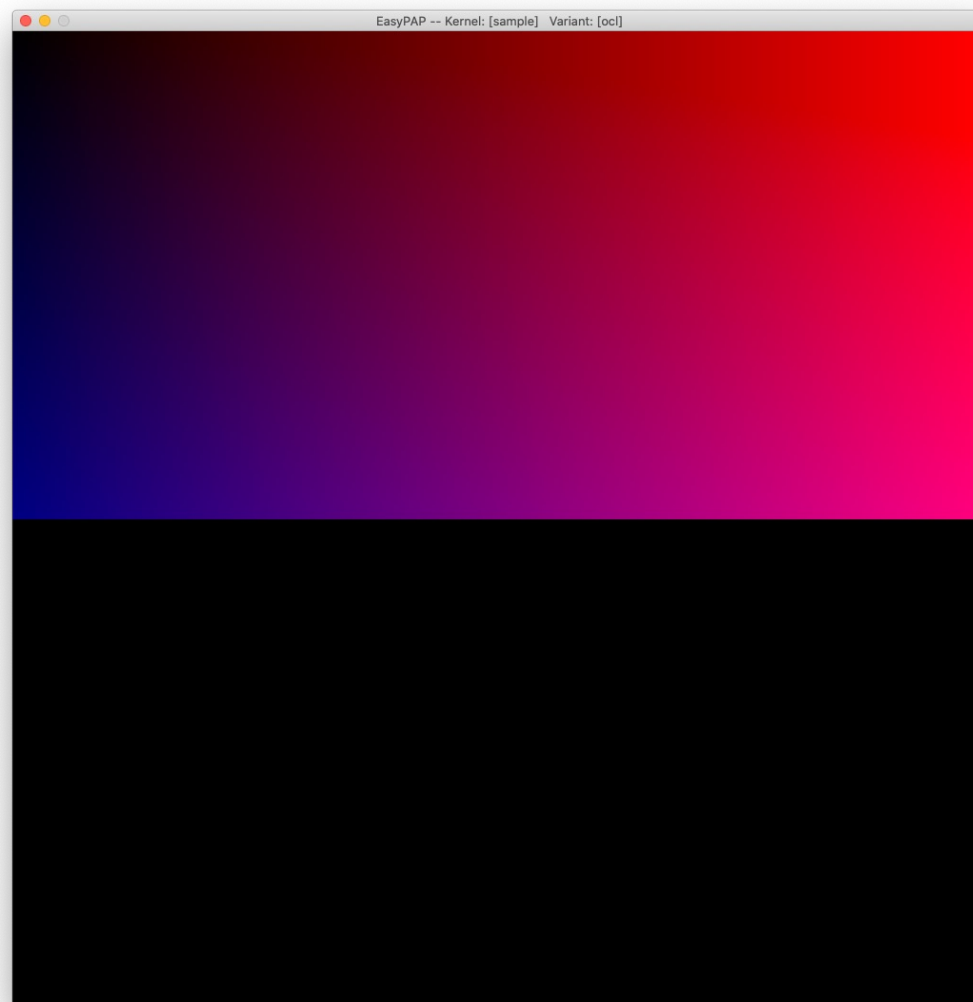
    for (unsigned it = 1; it <= nb_iter; it++) {
        // Set kernel arguments
        err = 0;
        err |= clSetKernelArg (compute_kernel, 0, sizeof (cl_mem), &cur_buffer);
        check (err, "Failed to set kernel arguments");

        err = clEnqueueNDRangeKernel (queue, compute_kernel, 2, NULL, global, local, 0, NULL, NULL);
        check (err, "Failed to execute kernel");
    }
    return 0;
}
```

OpenCL with EasyPAP

2D kernels

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



OpenCL with EasyPAP

2D kernels

- Let's fix our kernel to paint the whole image again
 - Each thread now computes 2 pixels
 - The following code does the job!

```
__kernel void sample_ocl (__global unsigned *img)
{
    int x = get_global_id (0);
    int y = get_global_id (1);
    unsigned color = 0xFF; // opacity = 100%
    color |= (x & 255) << 24; // the greater x, the more red we use
    color |= (y & 255) << 8; // the greater y, the more blue we use
    img [y * DIM + x] = color;

    // now address the lower half of image
    y += get_global_size (1); // y += 128 in our example
    color |= (y & 255) << 8; // blue
    img [y * DIM + x] = color;
}
```

Threads and Global Memory Access

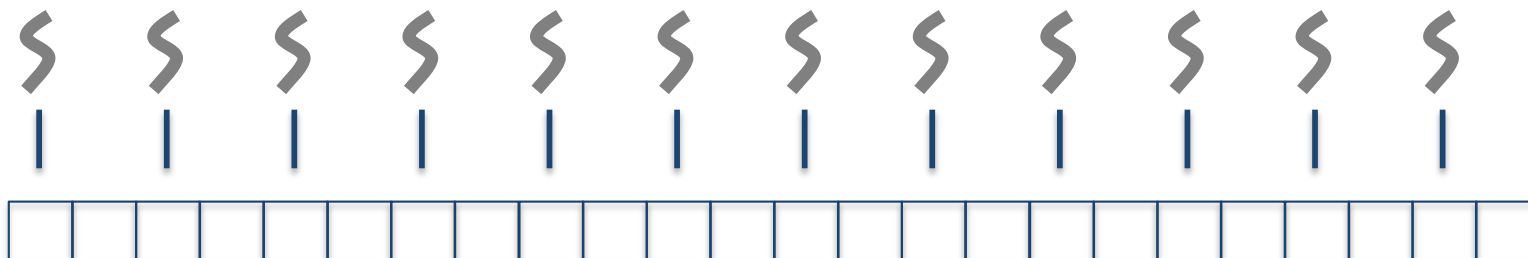
Memory concerns

OpenCL

- Coming back to our “ScalVec” kernel
 - Same config, except that we spawn **size-of-vector / 2** work items

```
__kernel void ScalVec(__global float *vec, float k)
{
    int index = get_global_id(0);

    vec[index*2] *= k;
    vec[index*2 + 1] *= k;
}
```



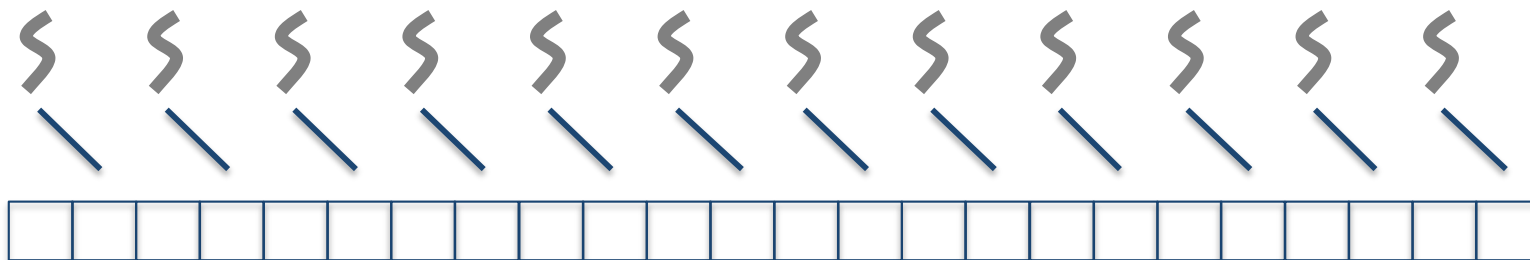
Memory concerns

OpenCL

- “ScalVec” kernel
 - Same config, except that we spawn **size-of-vector / 2** work items
 - **Performance is weak**

```
__kernel void ScalVec(__global float *vec, float k)
{
    int index = get_global_id(0);

    vec[index*2] *= k;
    vec[index*2 + 1] *= k;
}
```



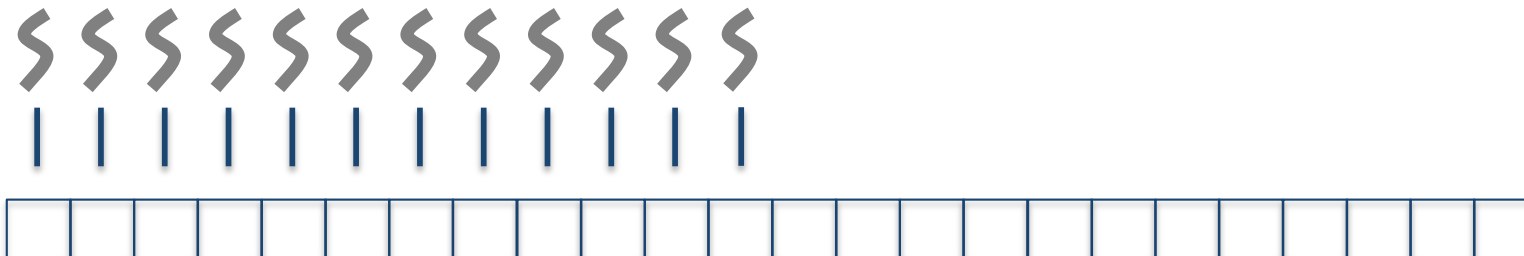
Memory concerns

OpenCL

- “ScalVec” kernel
 - Same config, except that we spawn **size-of-vector / 2** work items

```
__kernel void ScalVec(__global float *vec, float k)
{
    int index = get_global_id(0);
    int size = get_global_size(0); // #threads

    vec[index] *= k;
    vec[size + index] *= k;
}
```



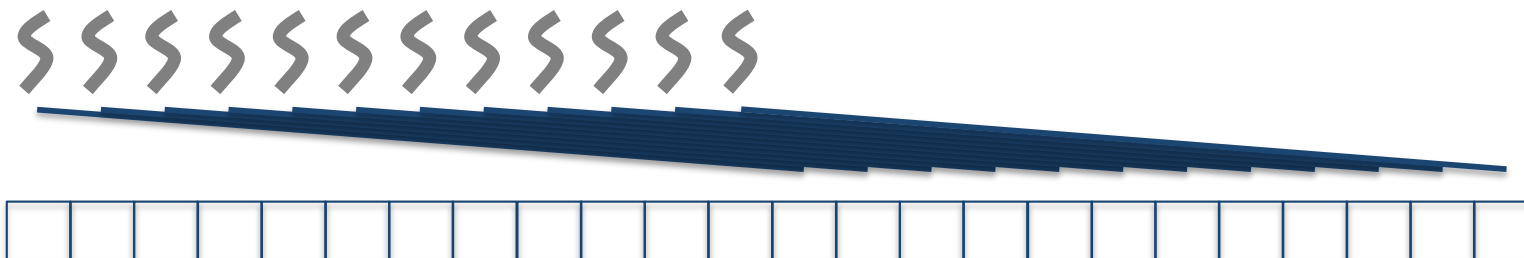
Memory concerns

OpenCL

- “ScalVec” kernel
 - Same config, except that we spawn **size-of-vector / 2** work items

```
__kernel void ScalVec(__global float *vec, float k)
{
    int index = get_global_id(0);
    int size = get_global_size(0); // #threads

    vec[index] *= k;
    vec[size + index] *= k;
}
```



Memory concerns

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 concerns

Memory Access Coalescing

- 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

```
__kernel void sample_ocl (__global unsigned *img)
{
    int x = get_global_id (0);
    int y = get_global_id (1);

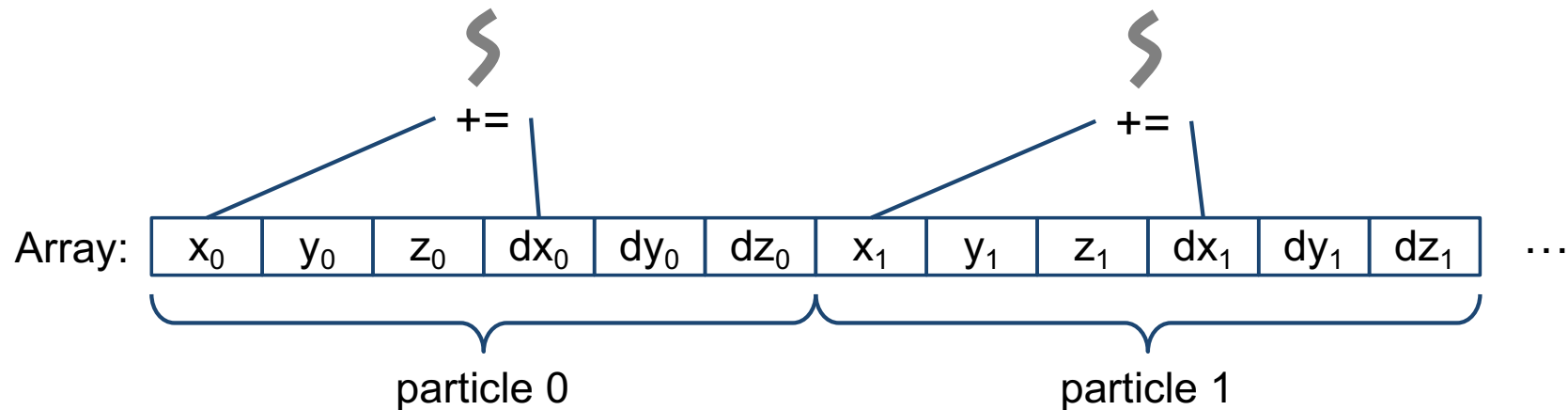
    unsigned color = 0xFFFF00FF; // yellow

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


Memory concerns

On the importance of data layout

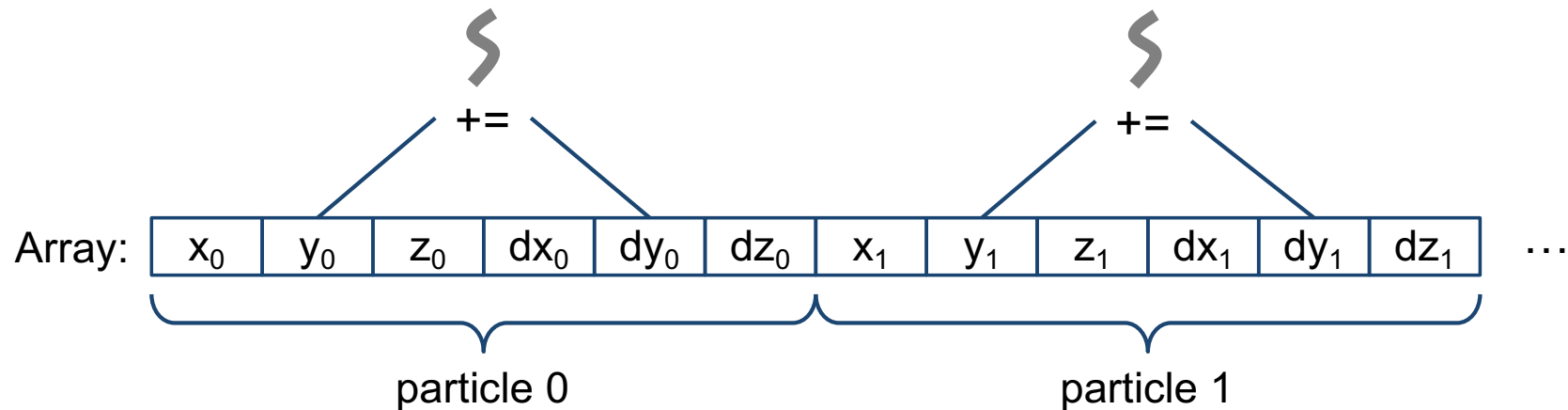
- Moving particles on a GPU
 - One thread per particle
 - $x += dx$



Memory concerns

On the importance of data layout

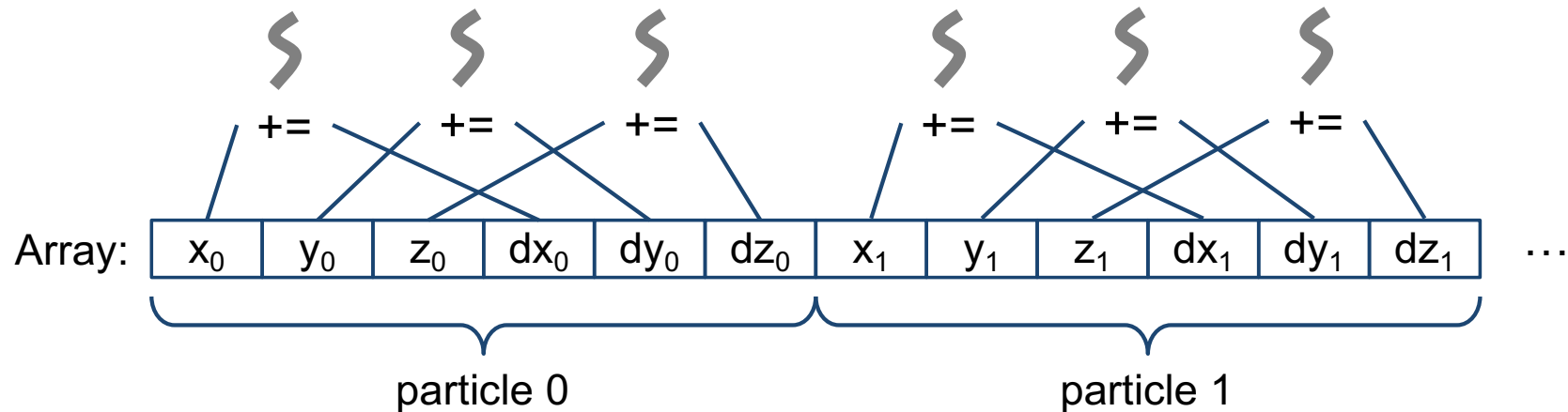
- Moving particles on a GPU
 - One thread per particle
 - $y += dy$



Memory concerns

On the importance of data layout

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

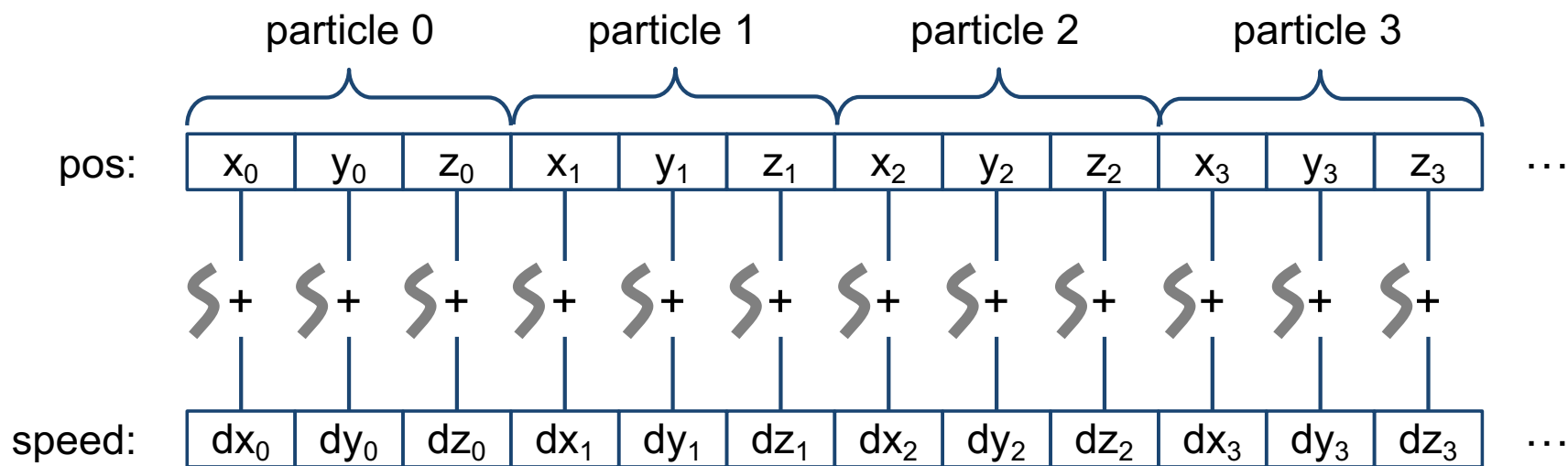


Memory concerns

On the importance of data layout

- 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 concerns

On the importance of data layout

- 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

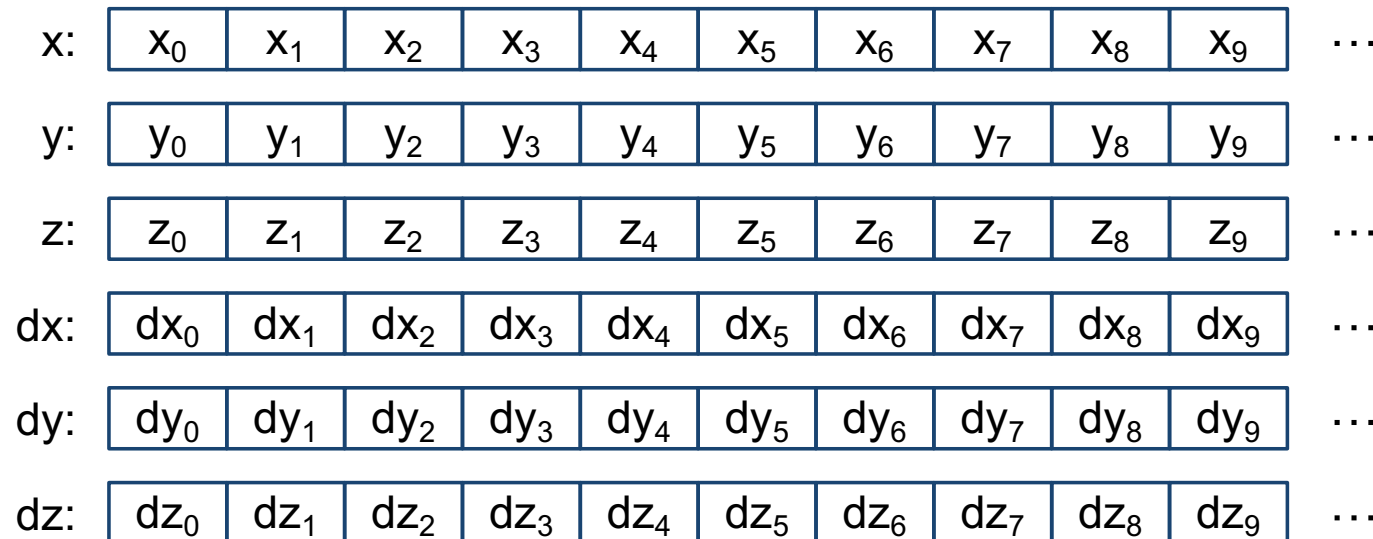
$$\sum_{\substack{0 \leq j < N \\ j \neq i}} f(d_{ij})$$

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

Memory concerns

On the importance of data layout

- The good solution is to opt for a “*Structure of Arrays*” (SoA) layout
 - Six arrays: x, y, z, dx, dy, dz



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

Memory concerns

2D kernels

- 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...

```
__kernel void transpose_ocl (__global unsigned *in,  
                             __global unsigned *out)  
{  
    int x = get_global_id (0);  
    int y = get_global_id (1);  
  
    out [x * DIM + y] = in [y * DIM + x];  
}
```

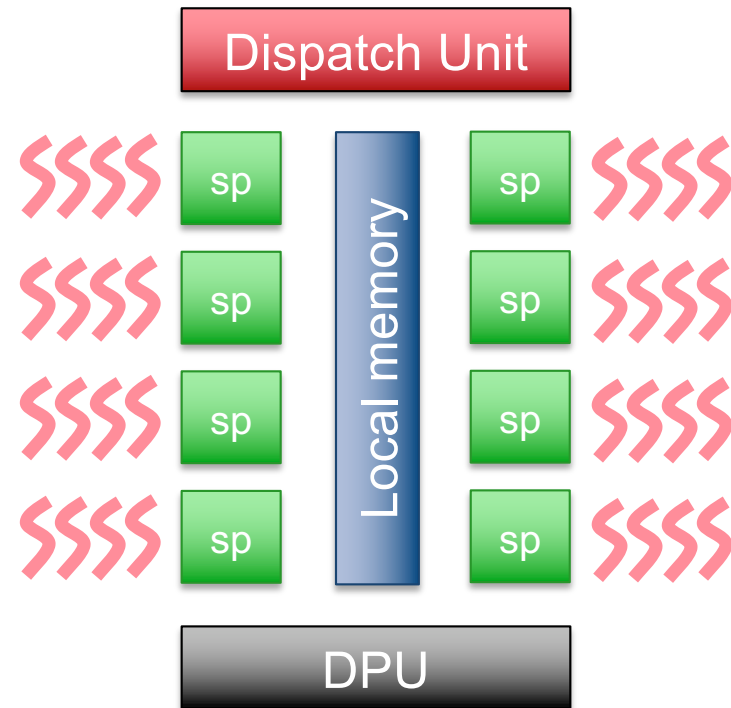
Thread divergence

Thread divergence

Illustration with the NVidia GPU architecture

- 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 (...
```

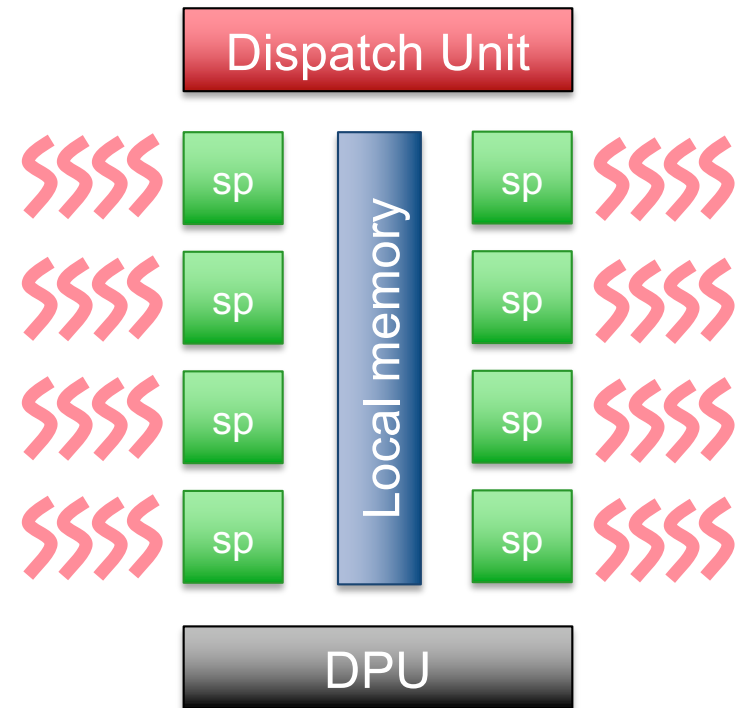


Thread divergence

Illustration with the NVidia GPU architecture

- What happens when threads execute a conditional instruction ?
- Threads belonging to the same warp cannot diverge

```
if (condition)
    do_this ();
else
    do_that ();
```

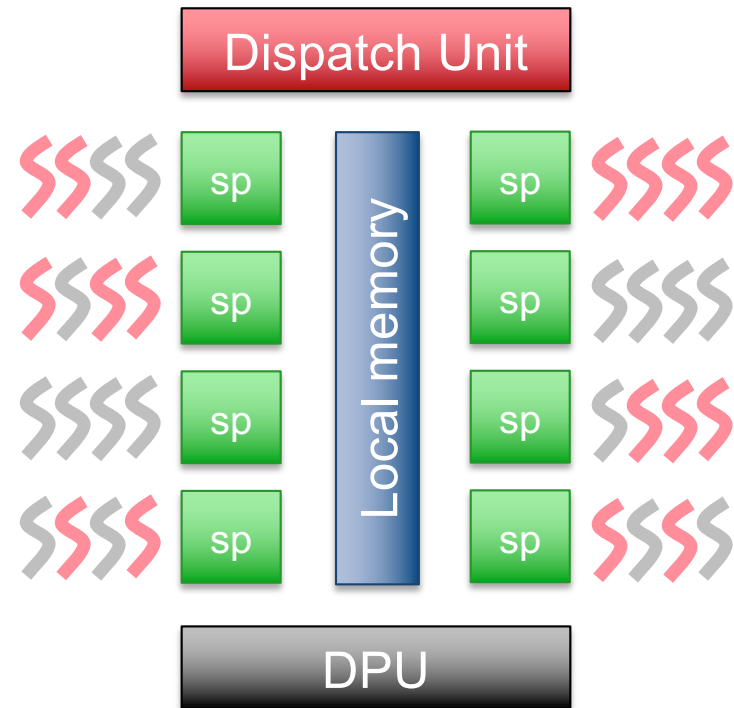


Thread divergence

Illustration with the NVidia GPU architecture

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

```
if (condition)
    do_this ();
else
    do_that ();
```

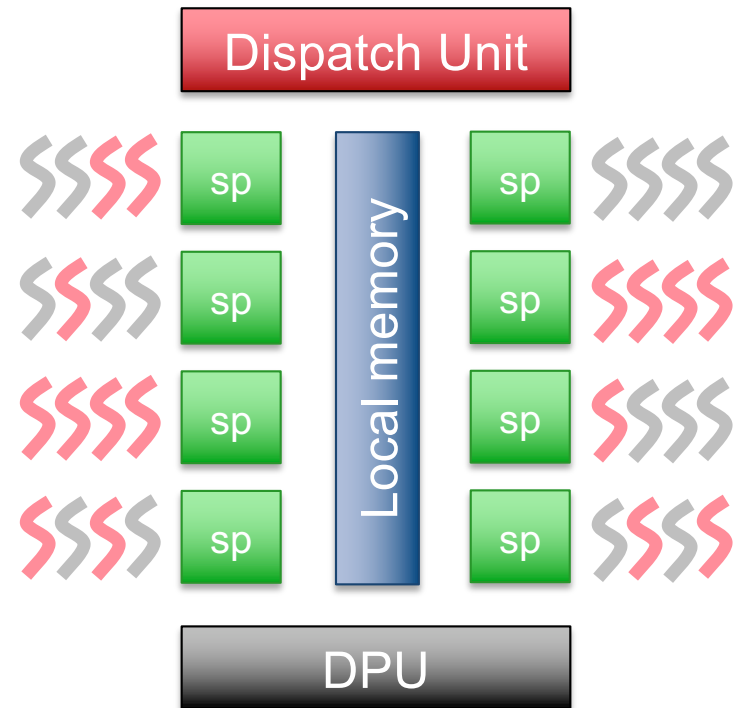


Thread divergence

Illustration with the NVidia GPU architecture

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

```
if (condition)
    do_this ();
else
    do_that ();
```



Thread divergence

Does it always hurt?

- Let us experiment to find out!
- Idea:
 - Bench a simple kernel with various divergence patterns
 - Perfect alternation



Thread divergence

Does it always hurt?

- Let us experiment to find out!
- Idea:
 - Bench a simple kernel with various divergence patterns

- Perfect alternation



- Two by two



Thread divergence

Does it always hurt?

- Let us experiment to find out!
- Idea:
 - Bench a simple kernel with various divergence patterns

- Perfect alternation



- Two by two



- Four by four



Thread divergence

Does it always hurt?

- Let us experiment to find out!
- Idea:
 - Bench a simple kernel with various divergence patterns

- Perfect alternation



- Two by two



- Four by four



- And so on...

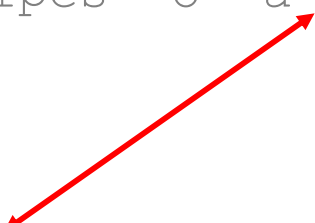


Thread divergence

Illustration with the NVidia GPU architecture

- 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 -o -a 3
```



```
unsigned mask = (1 << PARAM);  
if (x & mask)  
    out [y * DIM + x] = brighten (in [y * DIM + x]);  
else  
    out [y * DIM + x] = darken (in [y * DIM + x]);
```

Thread divergence

Illustration with the NVidia GPU architecture

x	x	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

Thread divergence

Illustration with the NVidia GPU architecture

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

Thread divergence

Illustration with the NVidia GPU architecture

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

Thread divergence

Illustration with the NVidia GPU architecture

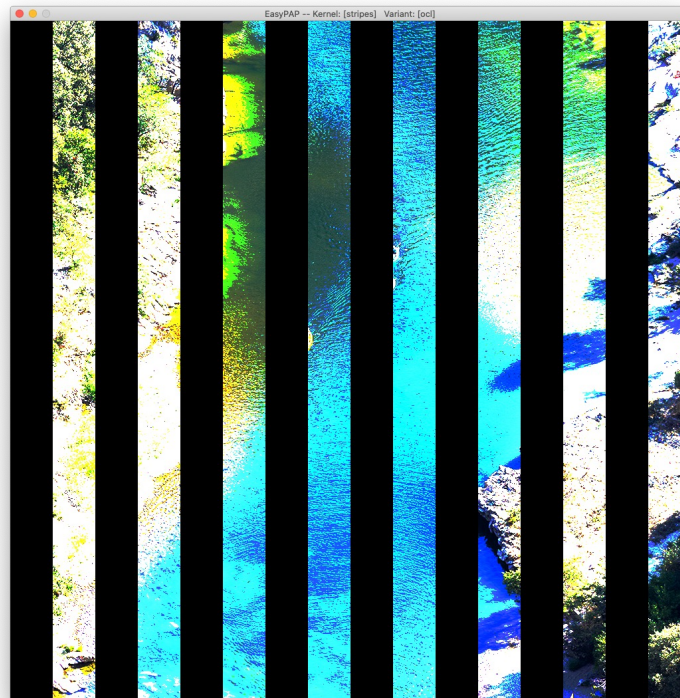
x	x	x & (1 << 3)
0	00000	00000
1	00001	00000
2	00010	00000
3	00011	00000
4	00100	00000
5	00101	00000
6	00110	00000
7	00111	00000
8	01000	01000
9	01001	01000
10	01010	01000
11	01011	01000
12	01100	01000
13	01101	01000
14	01110	01000
15	01111	01000

Thread divergence

Illustration with the NVidia GPU architecture

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

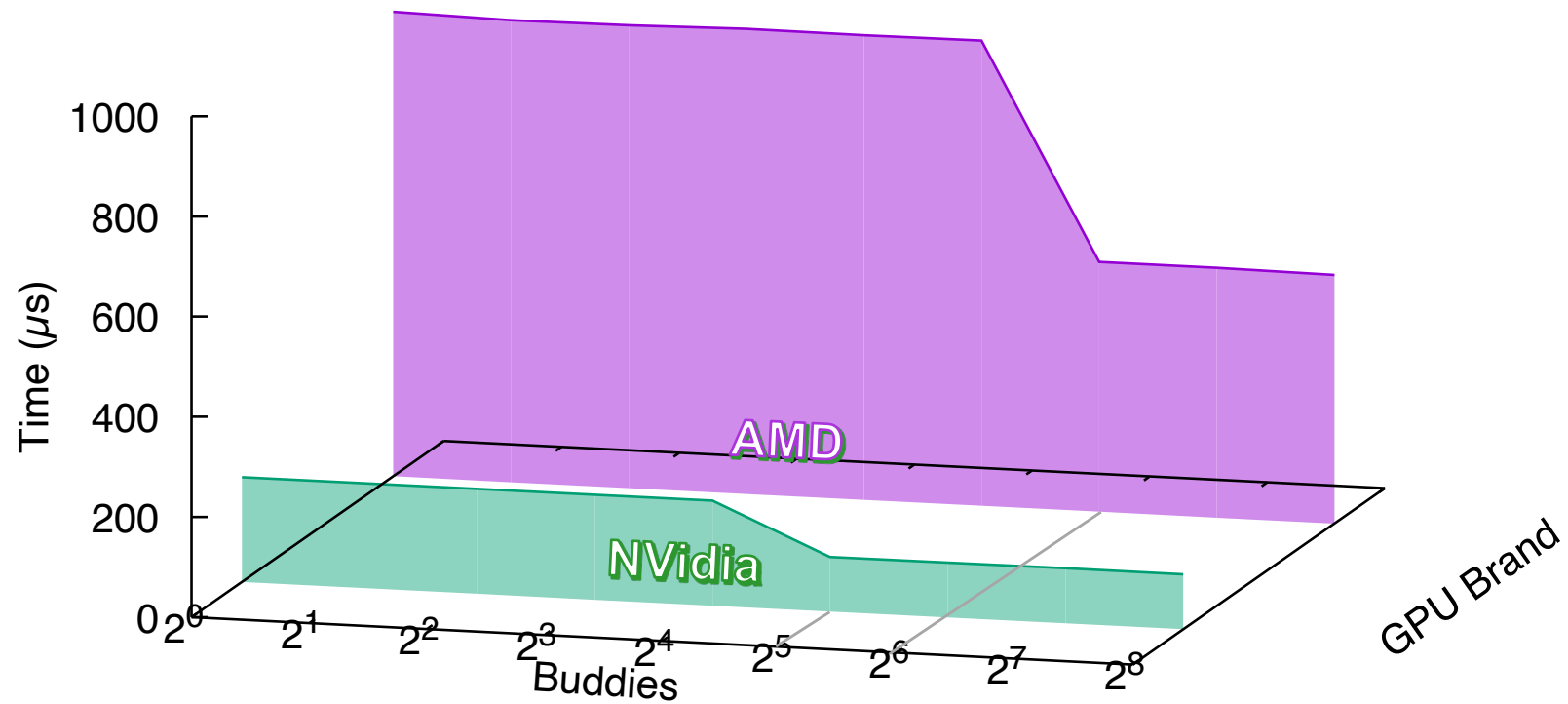
```
./run -l ... -k stripes -o -a 6
```



Thread Divergence

```
./run -k stripes -tw 64 -th 4 -o -i 1000 -n
```

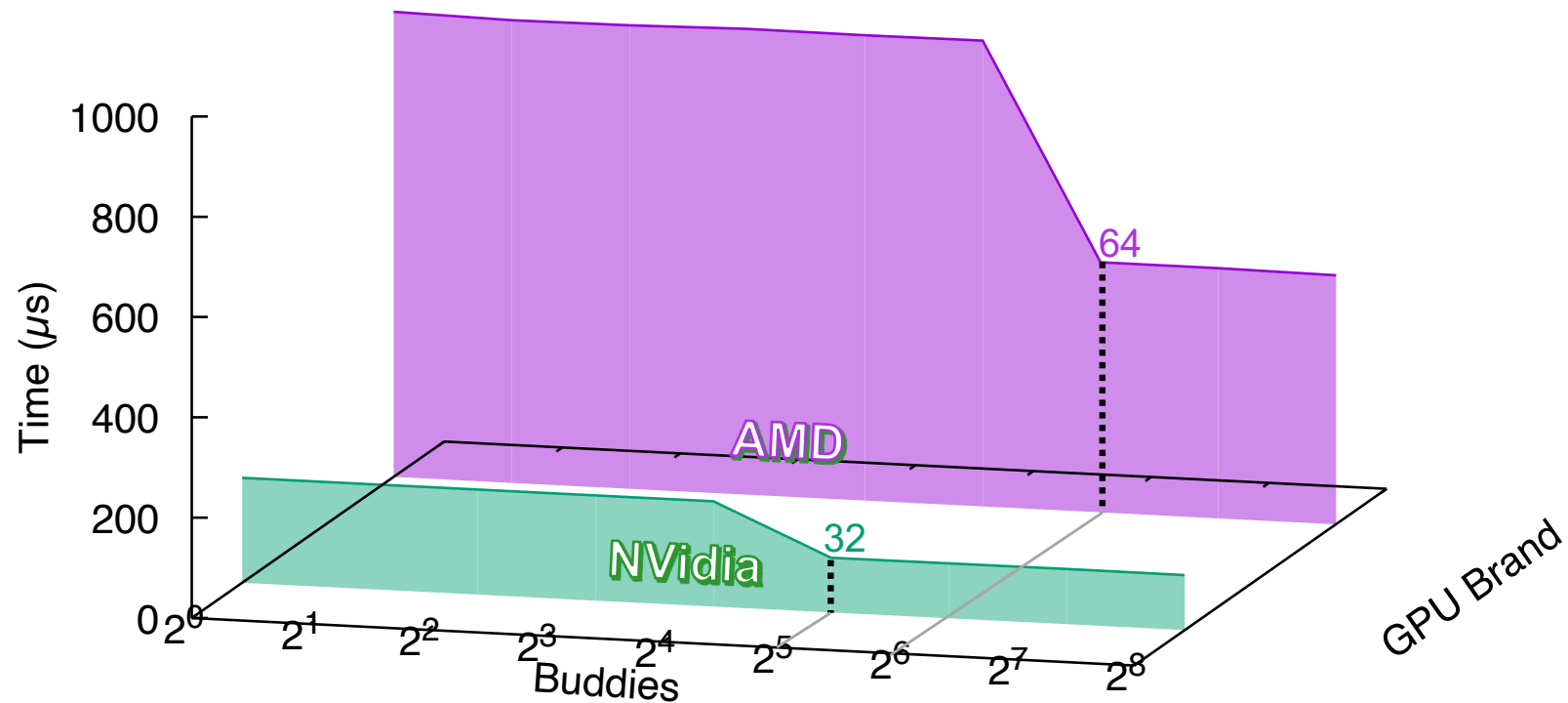
Radeon pro 560X 
GeForce RTX 2070 



Thread Divergence

```
./run -k stripes -tw 64 -th 4 -o -i 1000 -n
```

Radeon pro 560X 
GeForce RTX 2070 



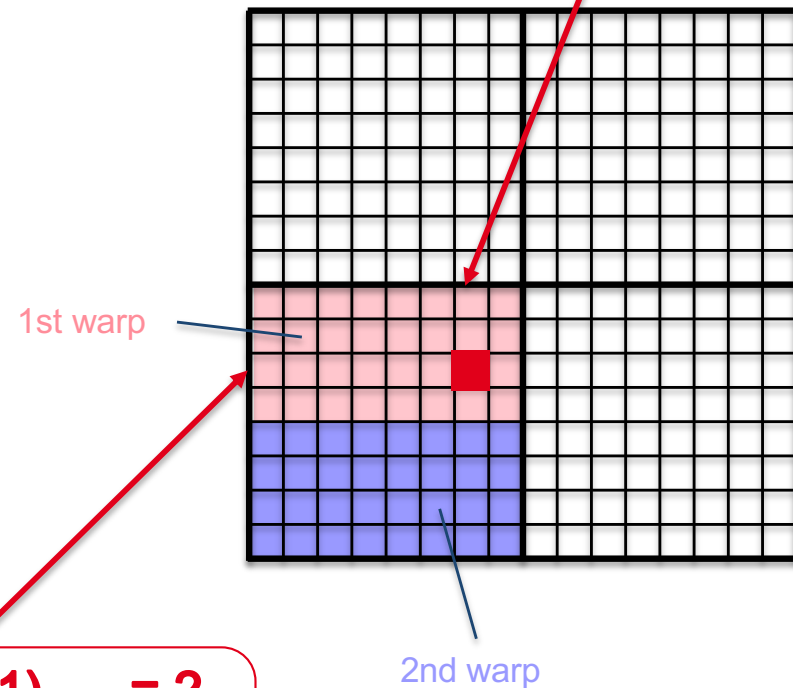
Only intra-warp divergence is harmful!

Workgroups and Shared Memory

OpenCL 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

`get_local_id(0) = 6`
`get_group_id(0) = 0`
`get_group_size(0) = 8`



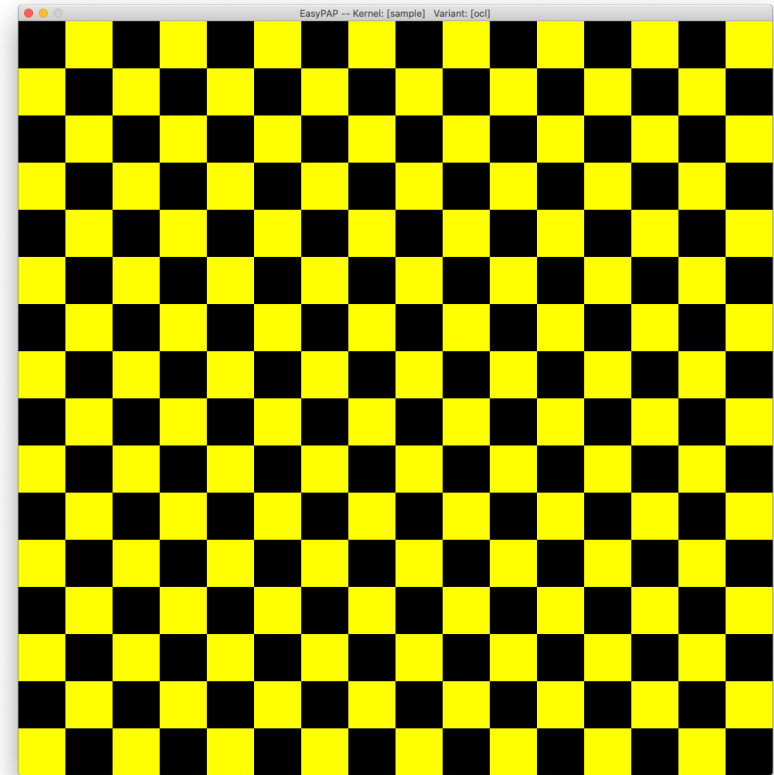
`get_local_id(1) = 2`
`get_group_id(1) = 1`
`get_group_size(1) = 8`

OpenCL Workgroups

```
./run -s 256 -k sample -o -tw 16 -th 16
```

```
__kernel void sample_ocl (__global unsigned *img)
{
    int x = get_global_id (0);
    int y = get_global_id (1);

    if ((get_group_id(0) + get_group_id(1)) % 2)
        img [y * DIM + x] = 0xFFFF00FF;
}
```

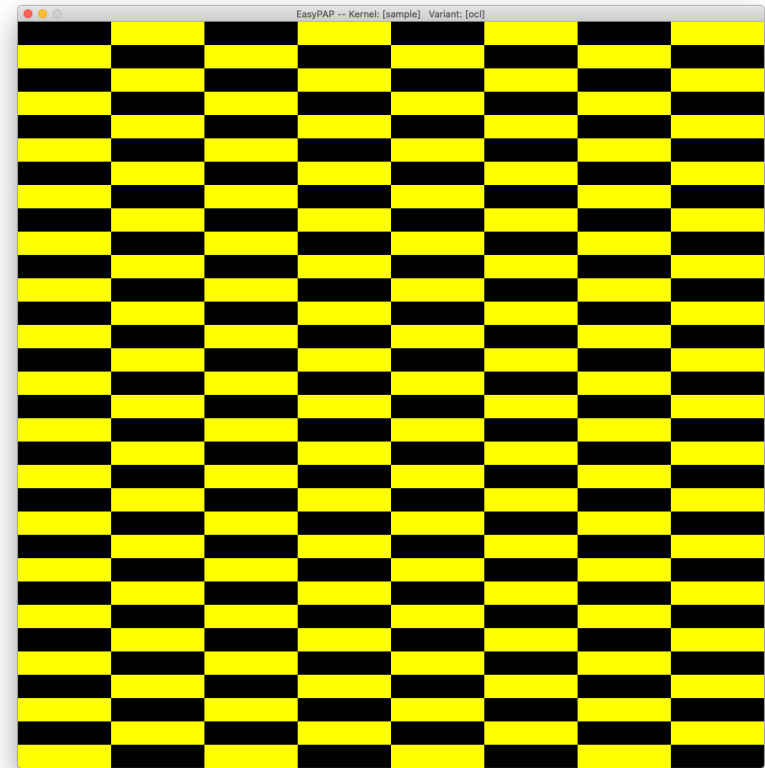


OpenCL Workgroups

```
./run -s 256 -k sample -o -tw 32 -th 8
```

```
__kernel void sample_ocl (__global unsigned *img)
{
    int x = get_global_id (0);
    int y = get_global_id (1);

    if ((get_group_id(0) + get_group_id(1)) % 2)
        img [y * DIM + x] = 0xFFFF00FF;
}
```

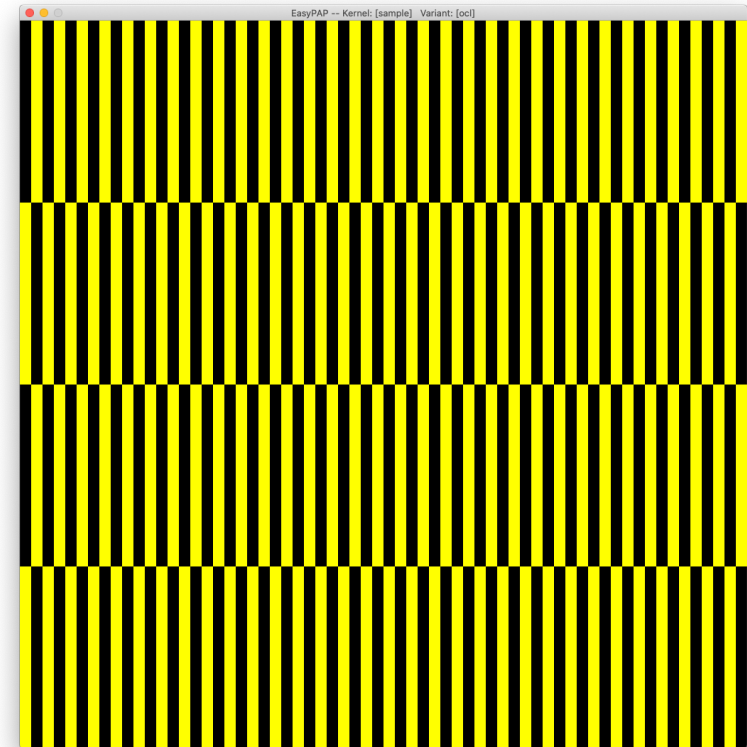


OpenCL Workgroups

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

```
__kernel void sample_ocl (__global unsigned *img)
{
    int x = get_global_id (0);
    int y = get_global_id (1);

    if ((get_group_id(0) + get_group_id(1)) % 2)
        img [y * DIM + x] = 0xFFFF00FF;
}
```



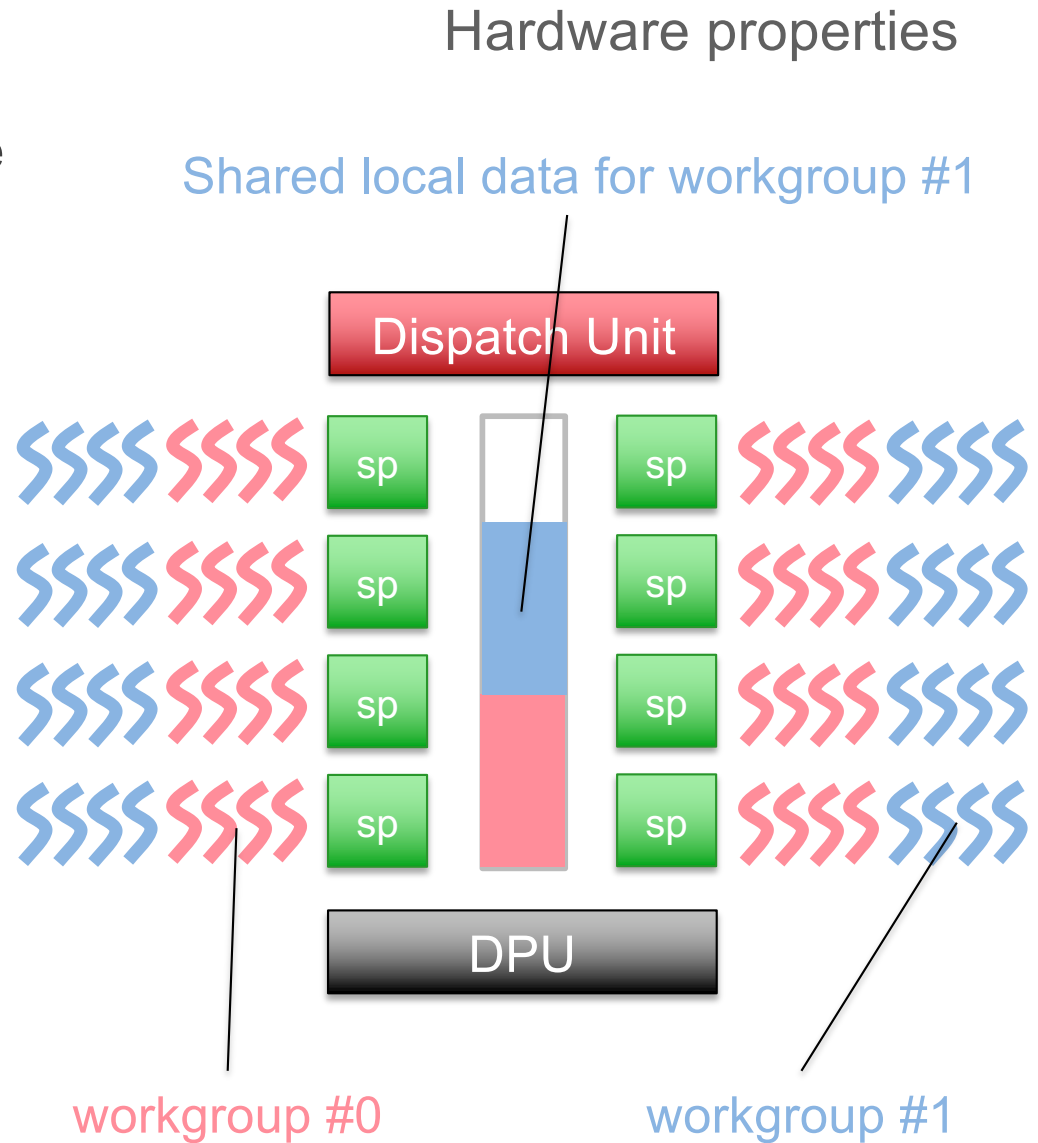
OpenCL Workgroups

Influence of workgroup shape on performance

- On a NVidia RTX 2070 card
- `./run -k sample -s 1024 -o -tw 16 -th 16 -i 1000 -n`
– 10.419
- `./run -k sample -s 1024 -o -tw 64 -th 4 -i 1000 -n`
– 11.878
- `./run -k sample -s 1024 -o -tw 4 -th 64 -i 1000 -n`
– **24.382**
– Reason: number of (uncoalesced) memory accesses

OpenCL Workgroups

- 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



OpenCL Workgroups

Sharing data through local memory

- Local memory is declared inside kernels using `__local`
- Example with this “oversimplified” `pixelize` kernel
 - Each workgroup has its private ‘`color`’ variable
 - Thread from the upper left corner sets this shared variable
 - Then threads synchronize to make sure ‘`color`’ has been written
 - Finally, all threads set their pixel to this color

```
__kernel void pixelize_ocl (__global unsigned *img)
{
    int x = get_global_id (0), y = get_global_id (1);
    int xloc = get_local_id (0), yloc = get_local_id (1);
    __local unsigned color;

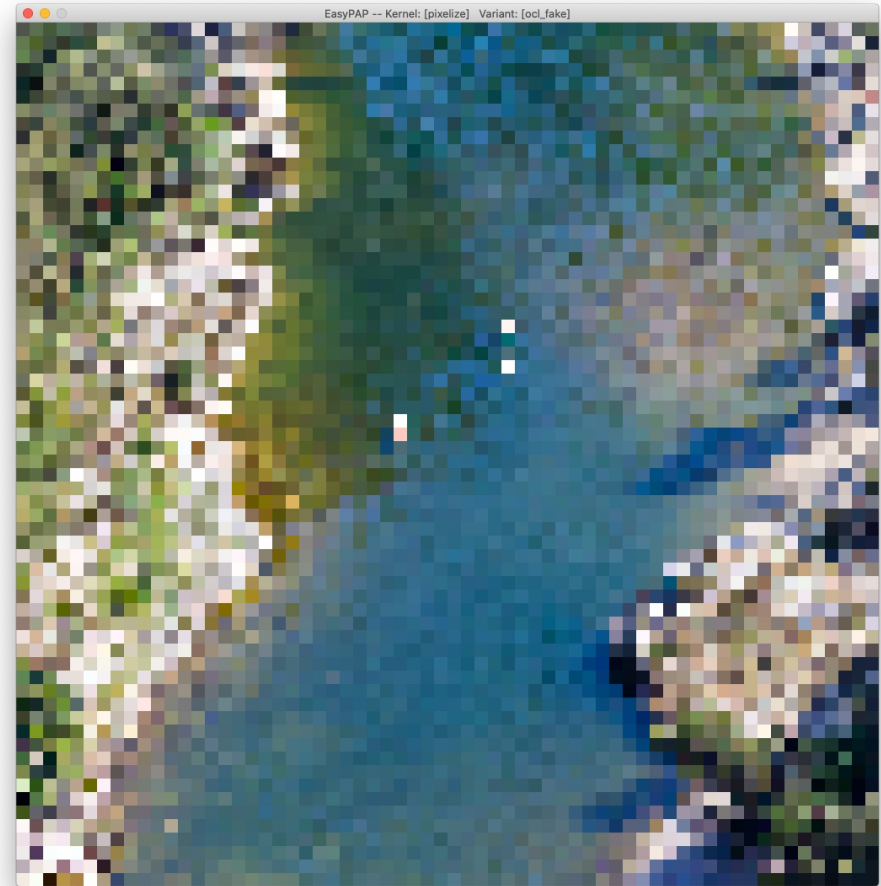
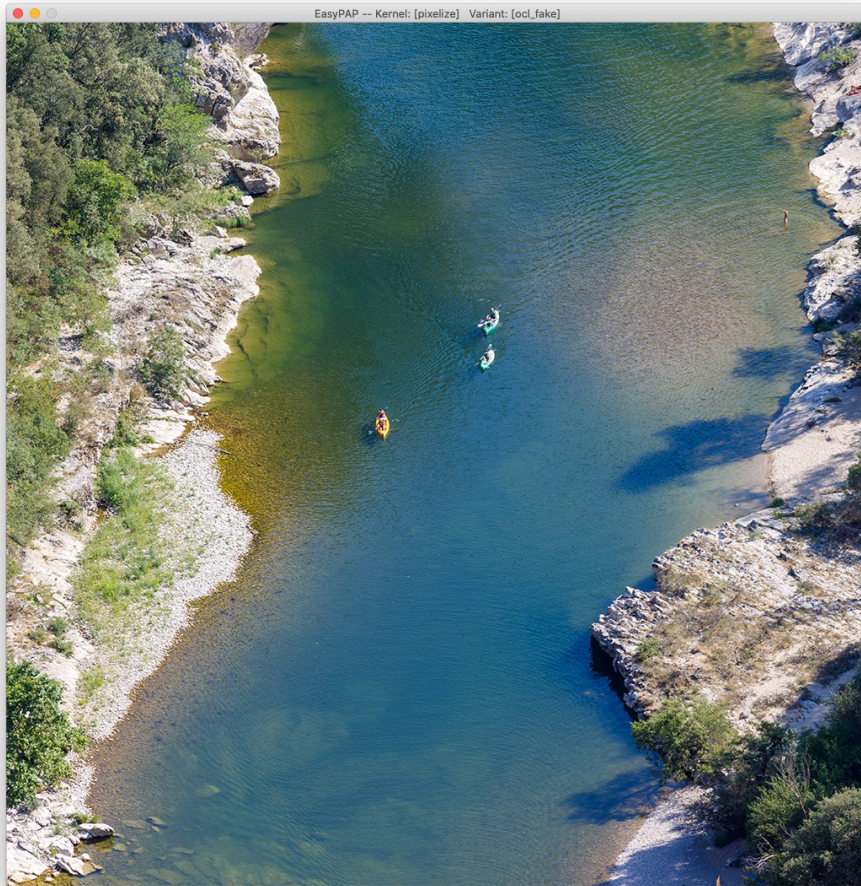
    if (xloc == 0 && yloc == 0) // upper left corner in each workgroup
        color = img [y * DIM + x]; // only one thread per wgrp reads from global memory

    barrier (CLK_LOCAL_MEM_FENCE);

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

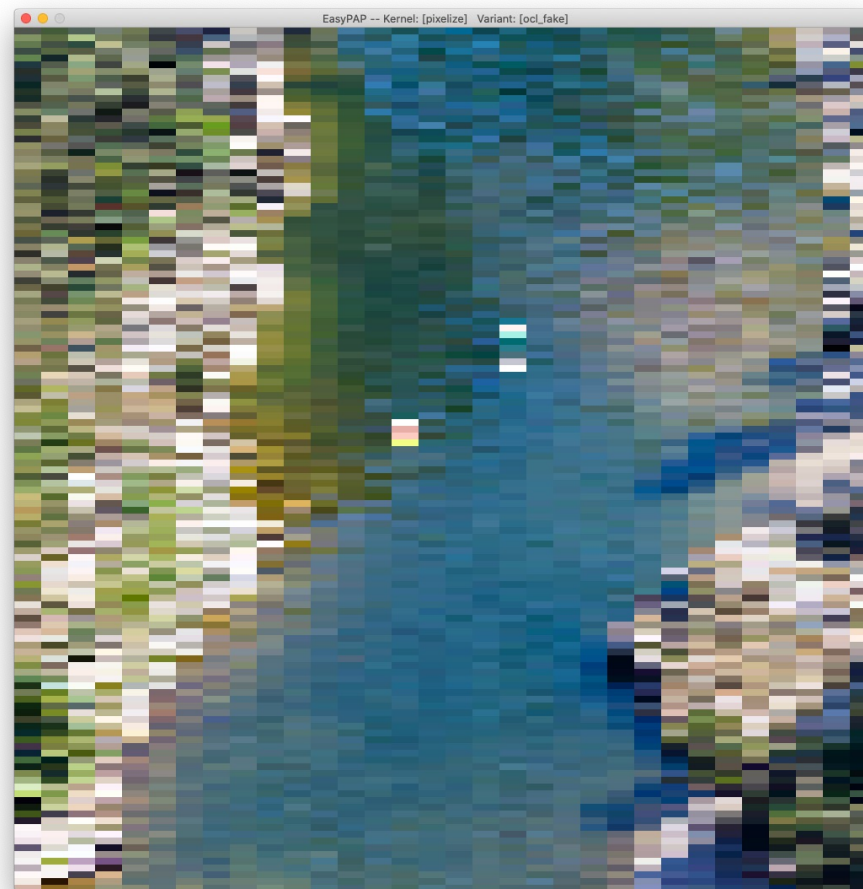
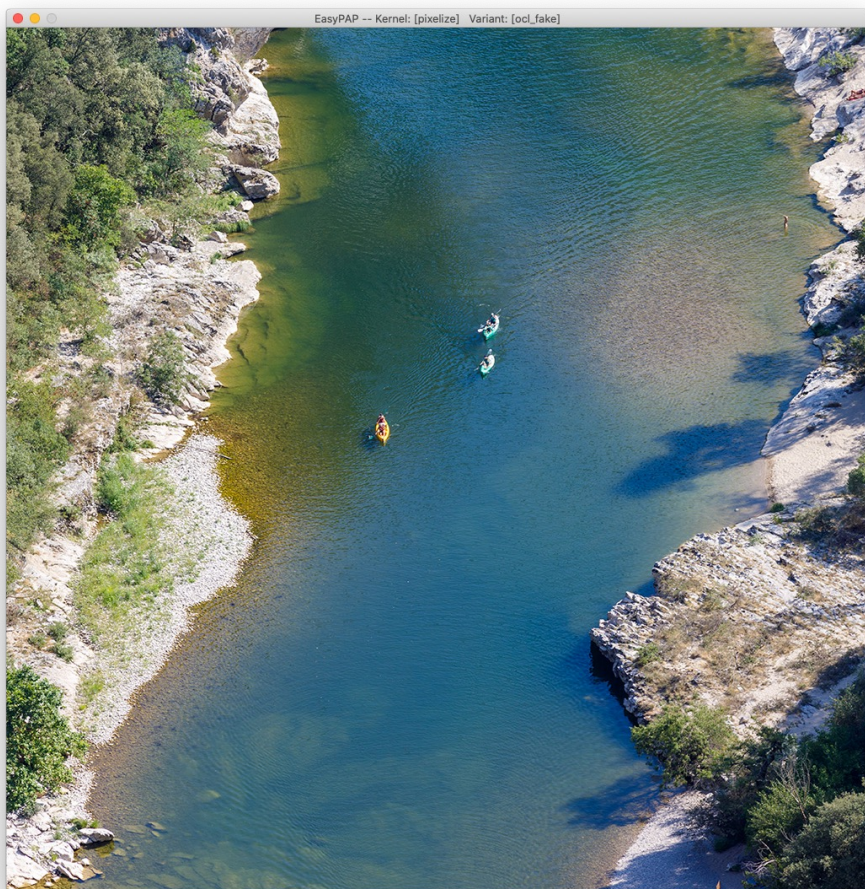
Sharing data through local memory

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



Sharing data through local memory

```
./run -l images/1024.png -k pixelize -g -tw 32 -th 8
```



Sharing data through local memory

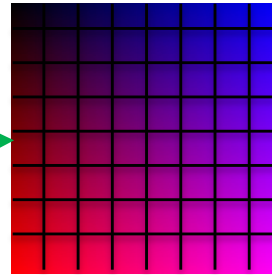
Tiling

- 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
- Let us use such "tiles" to solve our *transpose* problem!
 - Idea
 - 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

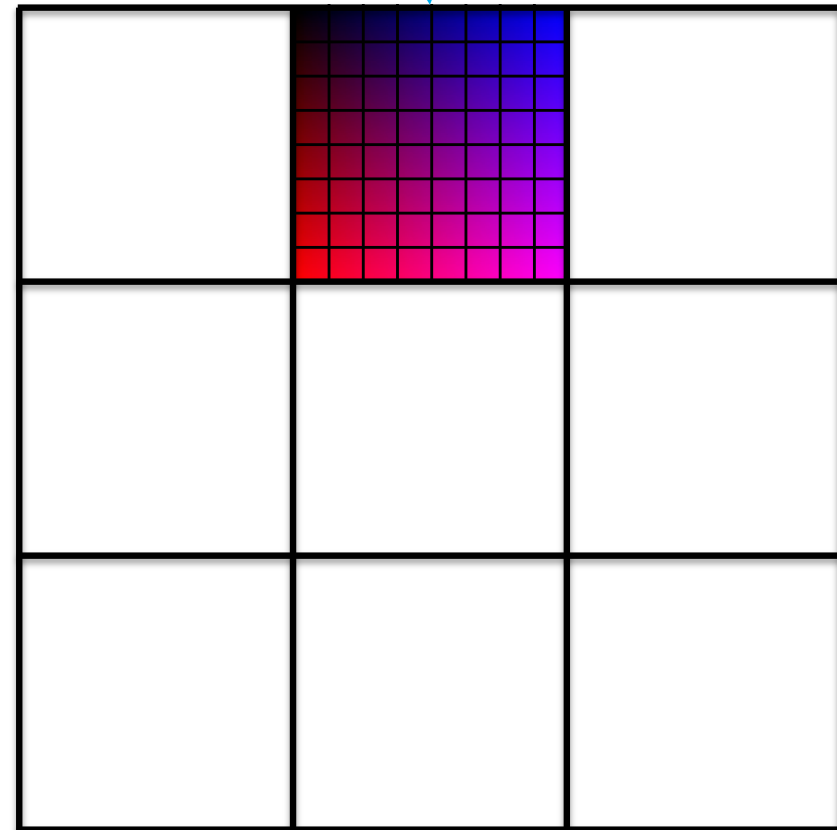
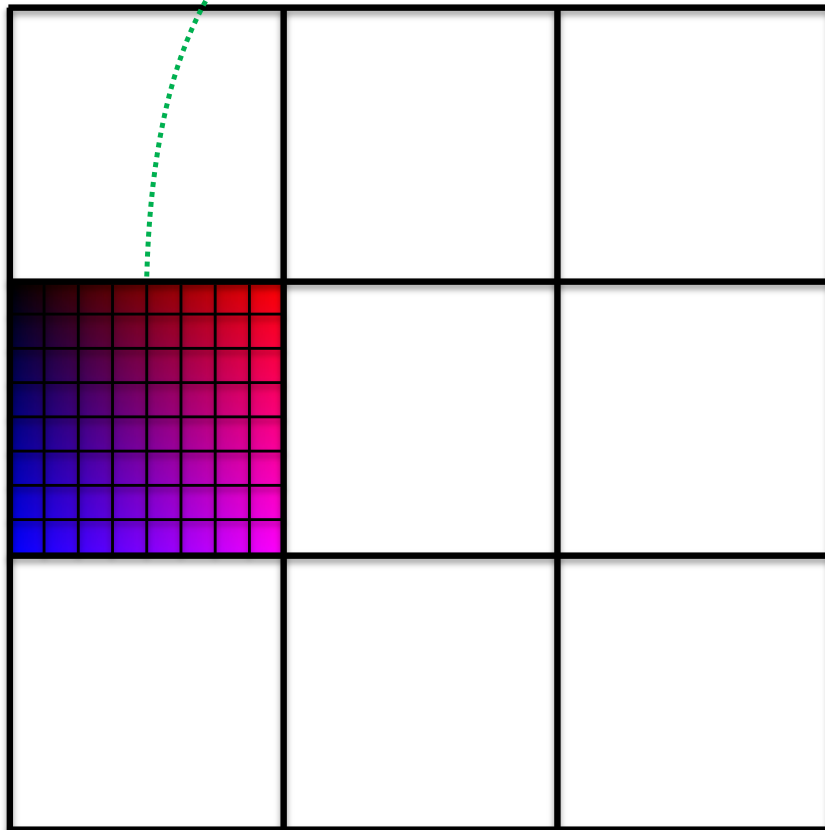
Sharing data through local memory

Tiled transpose

1. Read from A and write
"transposed data" into tile



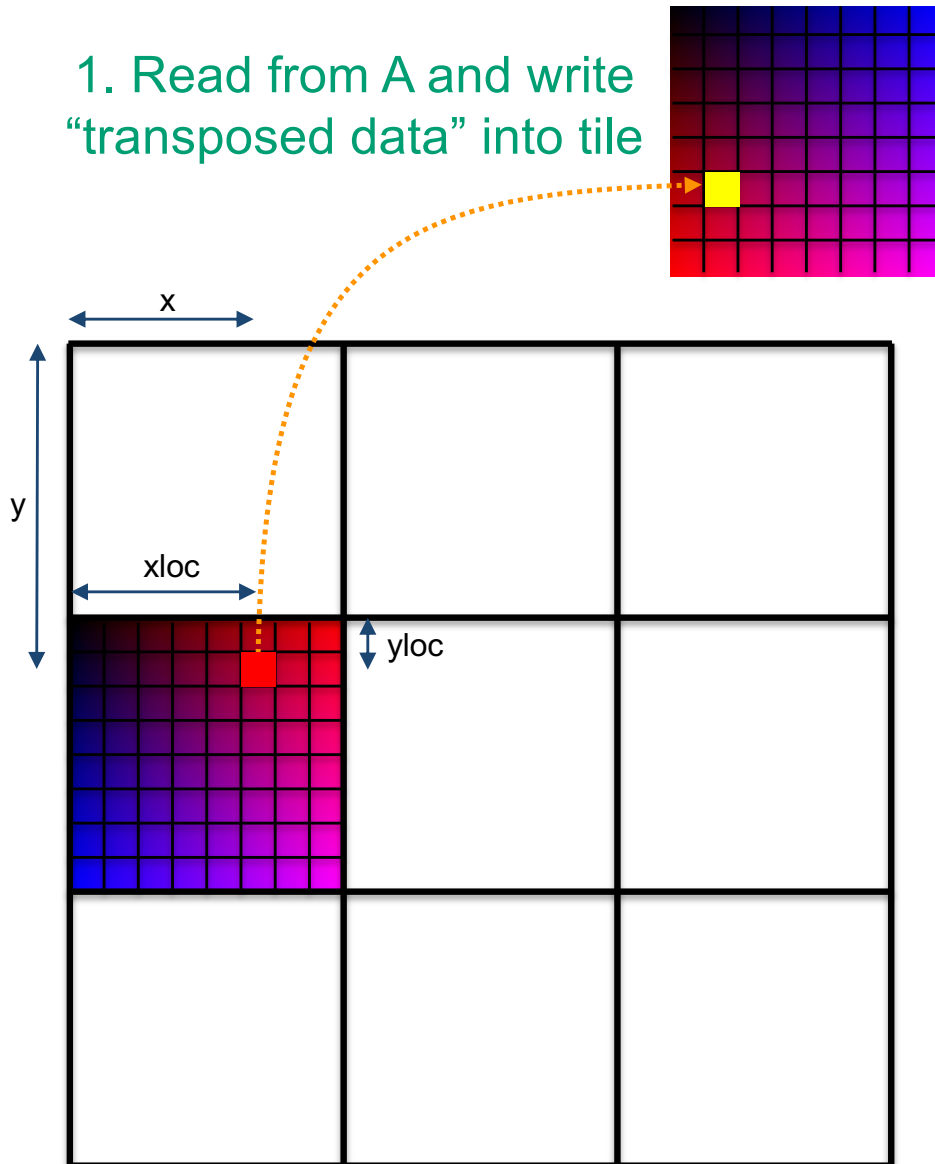
2. Memcpy tile to B



Sharing data through local memory

Tiled transpose : first step

1. Read from A and write
"transposed data" into tile



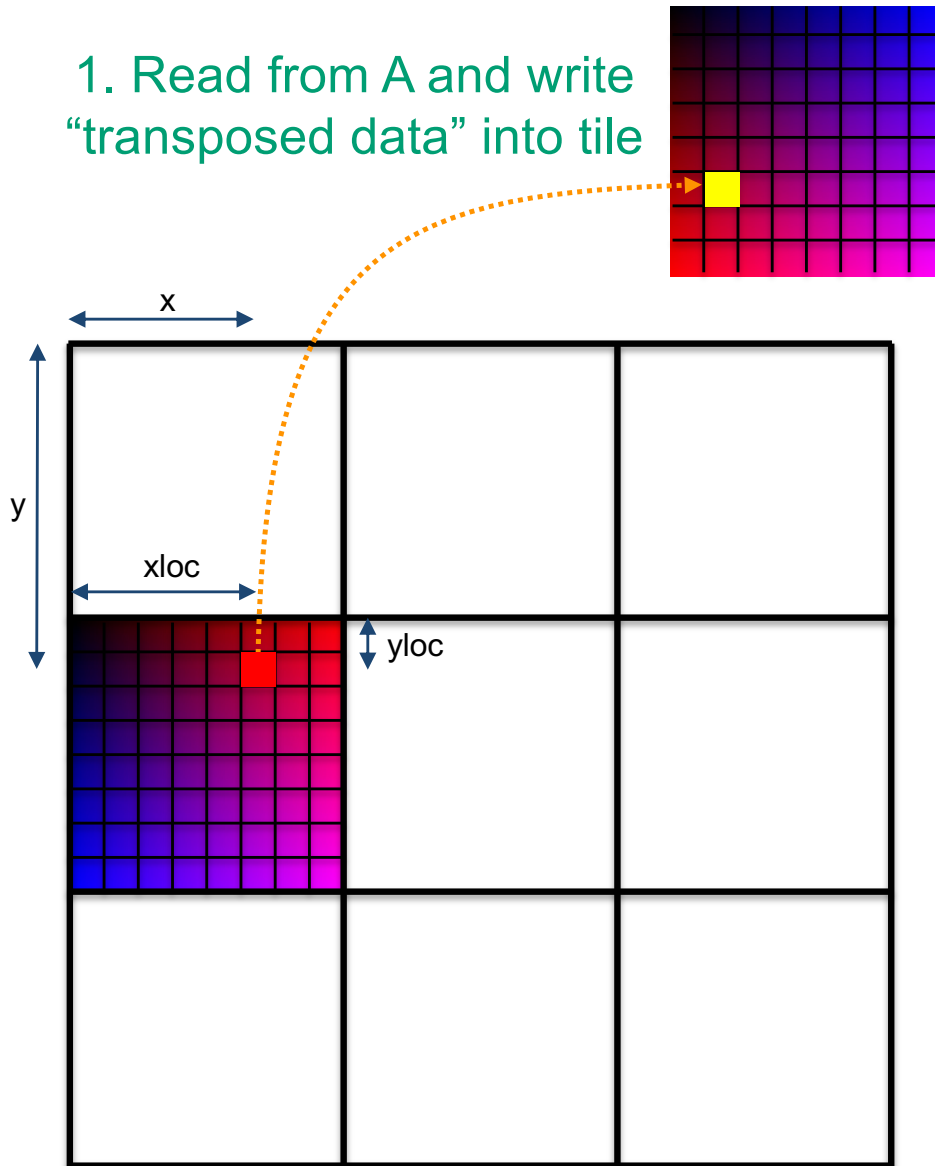
tile

```
__kernel void transpose_ocl (  
    __global unsigned *in,  
    __global unsigned *out)  
{  
    __local unsigned  
        tile [TILE_H][TILE_W];  
    int x = get_global_id (0);  
    int y = get_global_id (1);  
    int xloc = get_local_id (0);  
    int yloc = get_local_id (1);  
  
    tile [ ? ][ ? ] = in [y * DIM + x];  
}
```

Sharing data through local memory

Tiled transpose : first step

1. Read from A and write
“transposed data” into tile

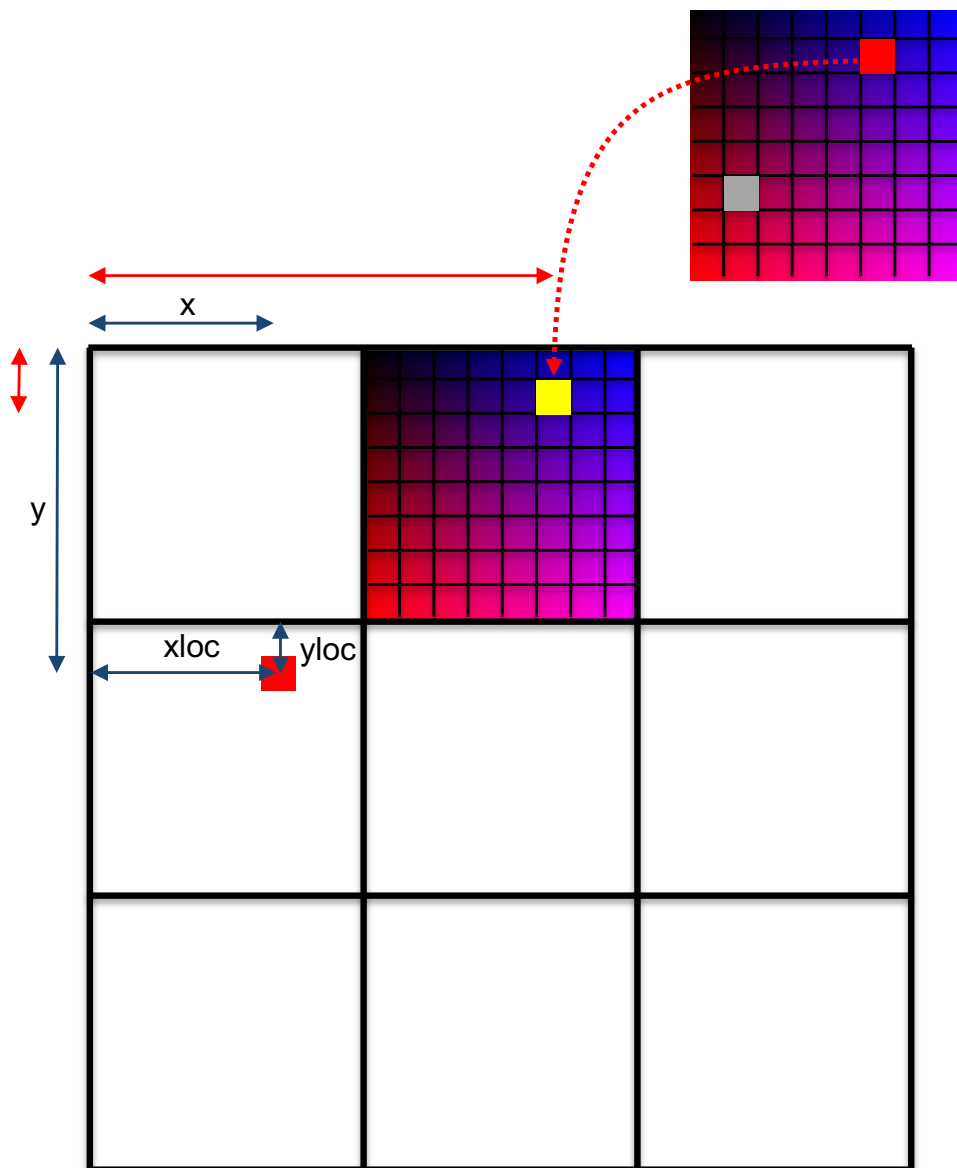


tile

```
__kernel void transpose_ocl (  
    __global unsigned *in,  
    __global unsigned *out)  
{  
    __local unsigned  
        tile [TILE_H][TILE_W];  
    int x = get_global_id (0);  
    int y = get_global_id (1);  
    int xloc = get_local_id (0);  
    int yloc = get_local_id (1);  
  
    tile [xloc][yloc] = in [y * DIM + x];  
}
```

Sharing data through local memory

Tiled transpose : first step



tile

```
__kernel void transpose_ocl (
    __global unsigned *in,
    __global unsigned *out)
{
    __local unsigned
        tile [TILE_H][TILE_W];

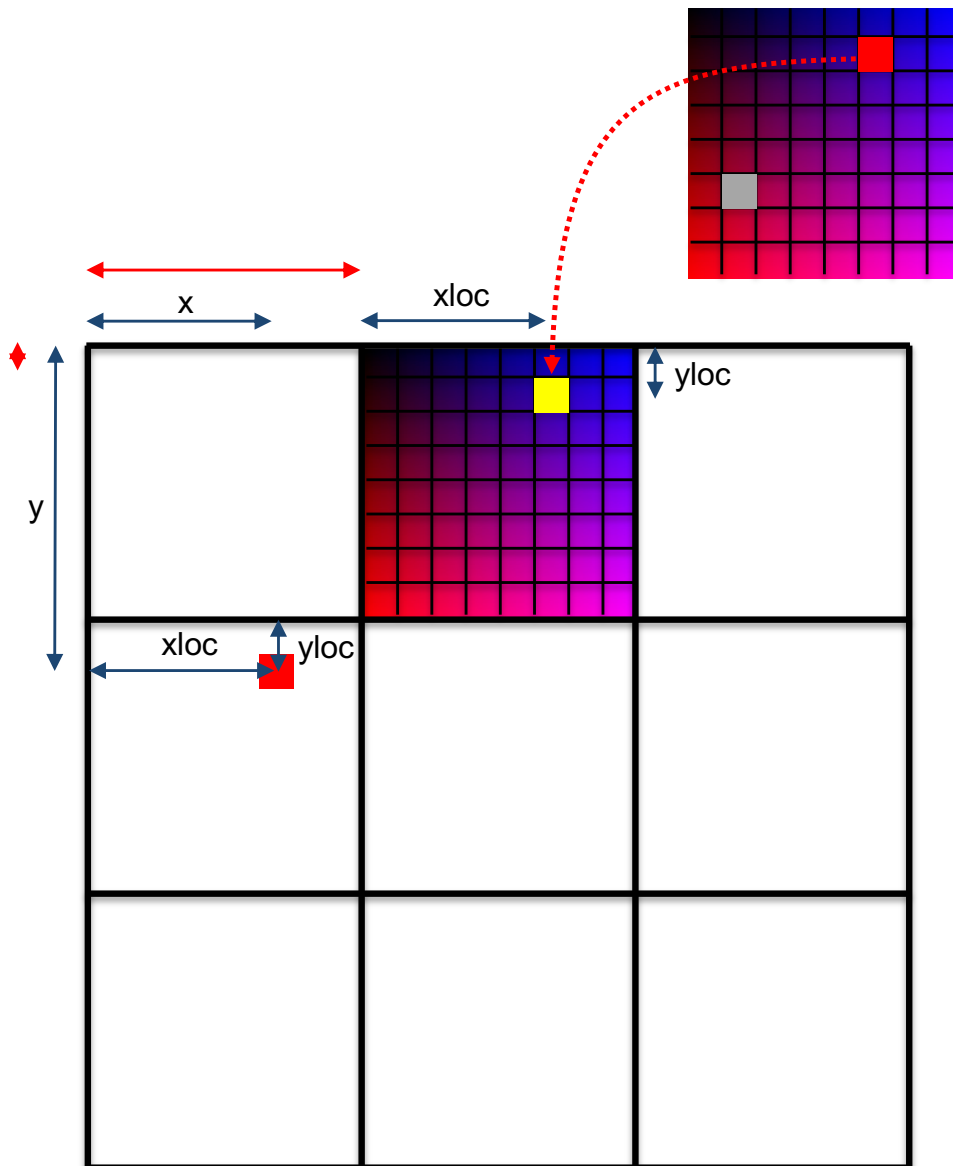
    int x = get_global_id (0);
    int y = get_global_id (1);
    int xloc = get_local_id (0);
    int yloc = get_local_id (1);

    tile [xloc][yloc] = in [y * DIM + x];

    out [ ? * DIM + ? ] =
        tile [yloc][xloc];
}
```

Sharing data through local memory

Tiled transpose : first step



tile

```
__kernel void transpose_ocl (
    __global unsigned *in,
    __global unsigned *out)
{
    __local unsigned
        tile [TILE_H][TILE_W];

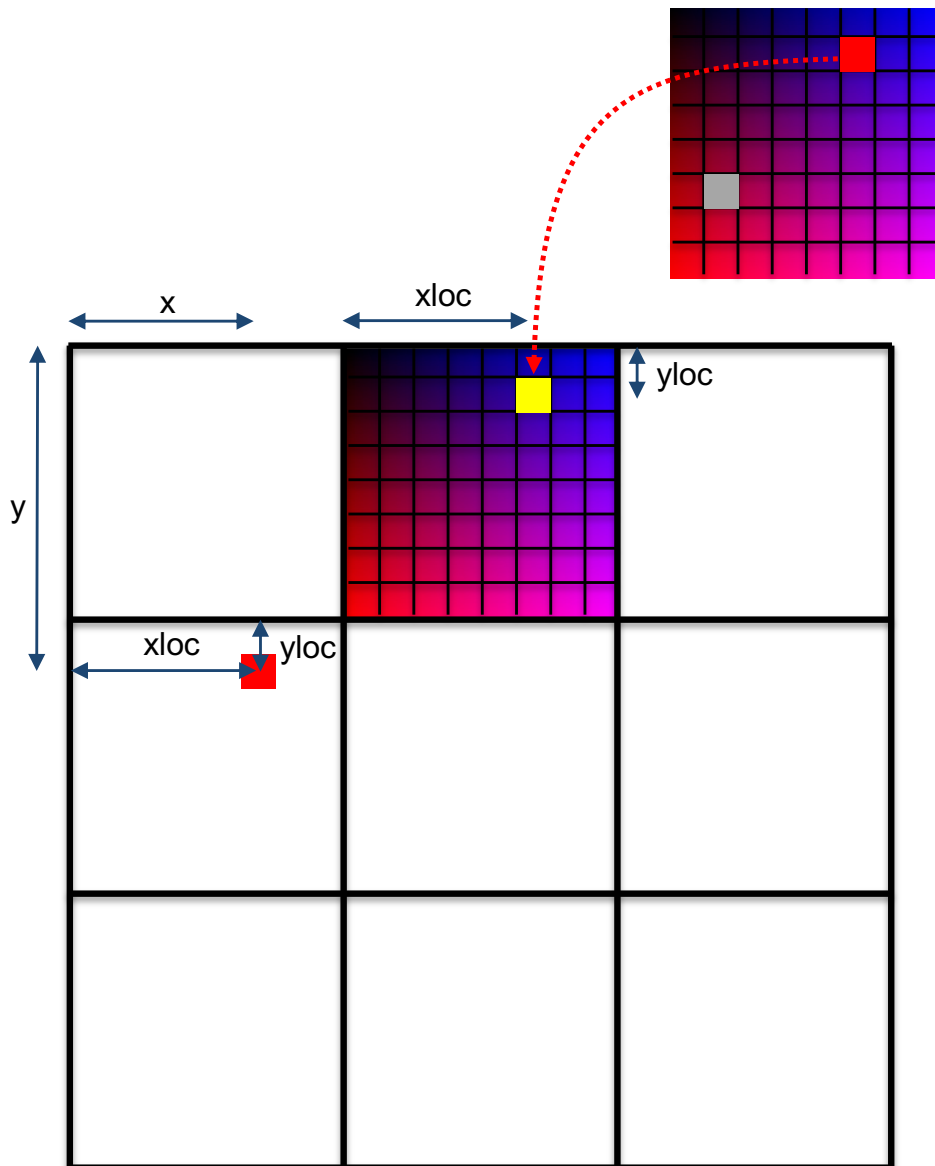
    int x = get_global_id (0);
    int y = get_global_id (1);
    int xloc = get_local_id (0);
    int yloc = get_local_id (1);

    tile [xloc][yloc] = in [y * DIM + x];

    out [( ? + yloc) * DIM +
        ? + xloc] =
        tile [yloc][xloc];
}
```

Sharing data through local memory

Tiled transpose : first step



tile

```
__kernel void transpose_ocl (  
    __global unsigned *in,  
    __global unsigned *out)  
{  
    __local unsigned  
        tile [TILE_H][TILE_W];  
    int x = get_global_id (0);  
    int y = get_global_id (1);  
    int xloc = get_local_id (0);  
    int yloc = get_local_id (1);  
  
    tile [xloc][yloc] = in [y * DIM + x];  
  
    out [(x - xloc + yloc) * DIM +  
        y - yloc + xloc] =  
        tile [yloc][xloc];  
}
```

Sharing data through local memory

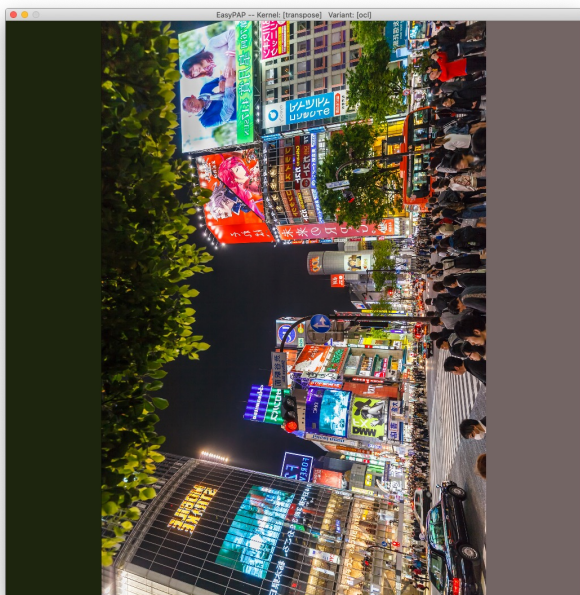
```
./run -g -k transpose ... -r 2
```



```
__kernel void transpose_ocl (  
    __global unsigned *in,  
    __global unsigned *out)  
{  
    __local unsigned  
        tile [TILE_H][TILE_W];  
    int x = get_global_id (0);  
    int y = get_global_id (1);  
    int xloc = get_local_id (0);  
    int yloc = get_local_id (1);  
  
    tile [xloc][yloc] = in [y * DIM + x];  
  
    out [(x - xloc + yloc) * DIM +  
        y - yloc + xloc] =  
        tile [yloc][xloc];  
}
```


Sharing data through local memory

That's better!



```
__kernel void transpose_ocl (
    __global unsigned *in,
    __global unsigned *out)
{
    __local unsigned
        tile [TILE_H][TILE_W];
    int x = get_global_id (0);
    int y = get_global_id (1);
    int xloc = get_local_id (0);
    int yloc = get_local_id (1);

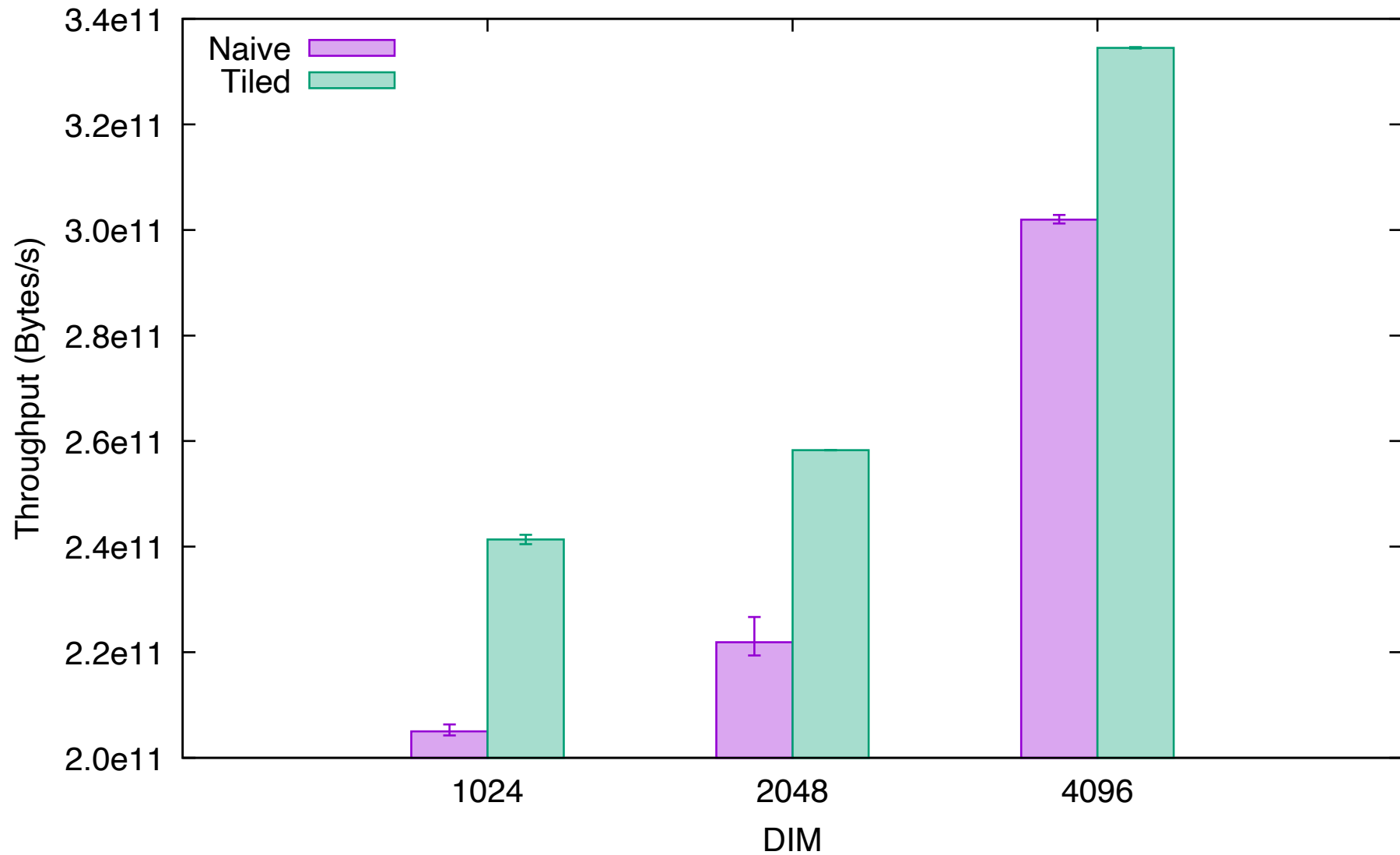
    tile [xloc][yloc] = in [y * DIM + x];

    barrier (CLK_LOCAL_MEM_FENCE);

    out [(x - xloc + yloc) * DIM +
        y - yloc + xloc] =
        tile [yloc][xloc];
}
```

Matrix transpose on GeForce GTX 2080

```
./run -g -k transpose -i 1000 -n -s <dim>
```



How about a magic trick?

Just for fun

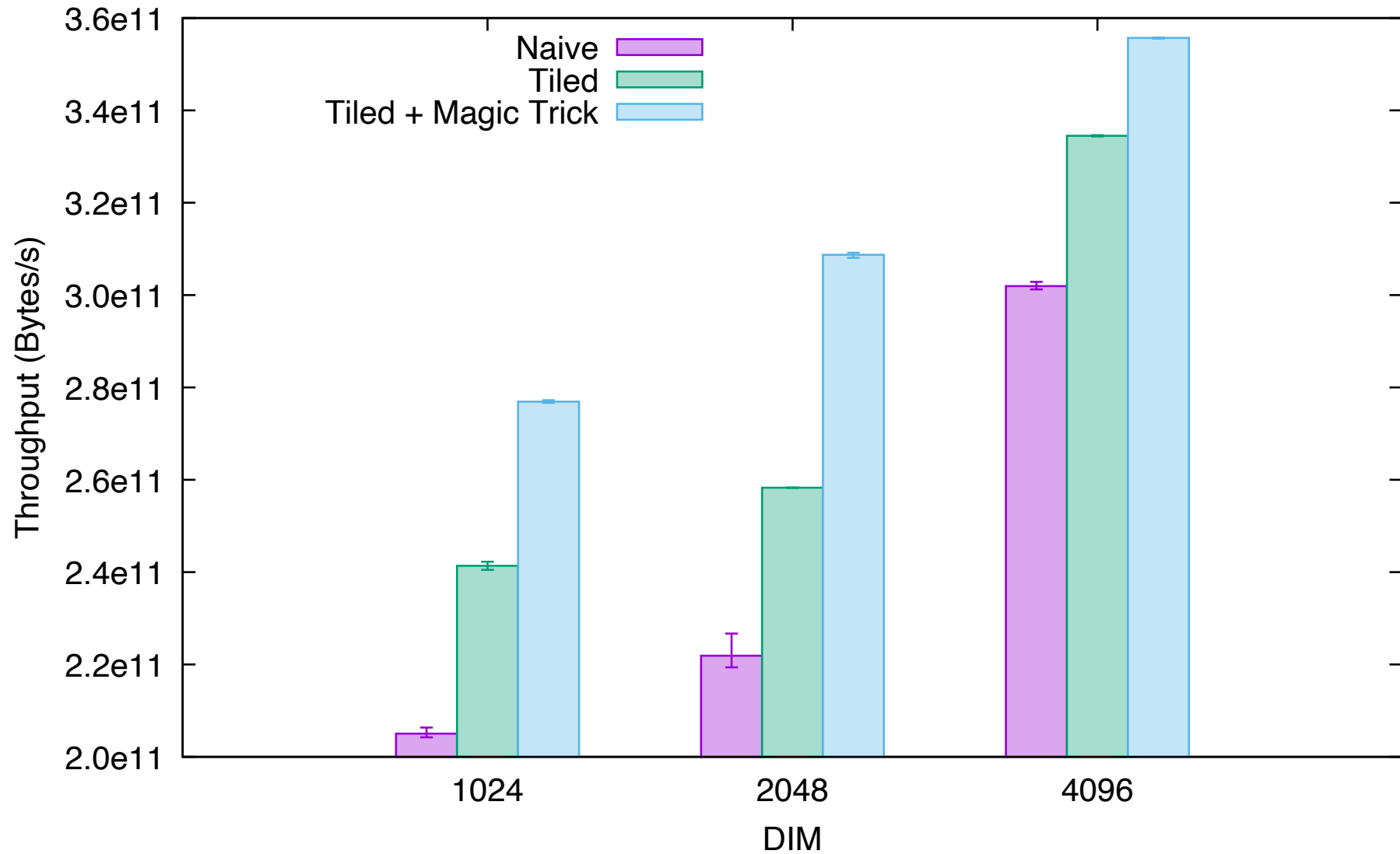


Why the hell do we add
this extra column
we don't even use?!

```
__kernel void transpose_ocl (  
    __global unsigned *in,  
    __global unsigned *out)  
{  
    __local unsigned tile [TILE_H][TILE_W+1];  
    int x = get_global_id (0);  
    int y = get_global_id (1);  
    int xloc = get_local_id (0);  
    int yloc = get_local_id (1);  
  
    tile [xloc][yloc] = in [y * DIM + x];  
  
    barrier (CLK_LOCAL_MEM_FENCE);  
  
    out [(x - xloc + yloc) * DIM +  
        y - yloc + xloc] =  
        tile [yloc][xloc];  
}
```

Tadaaaaaam!

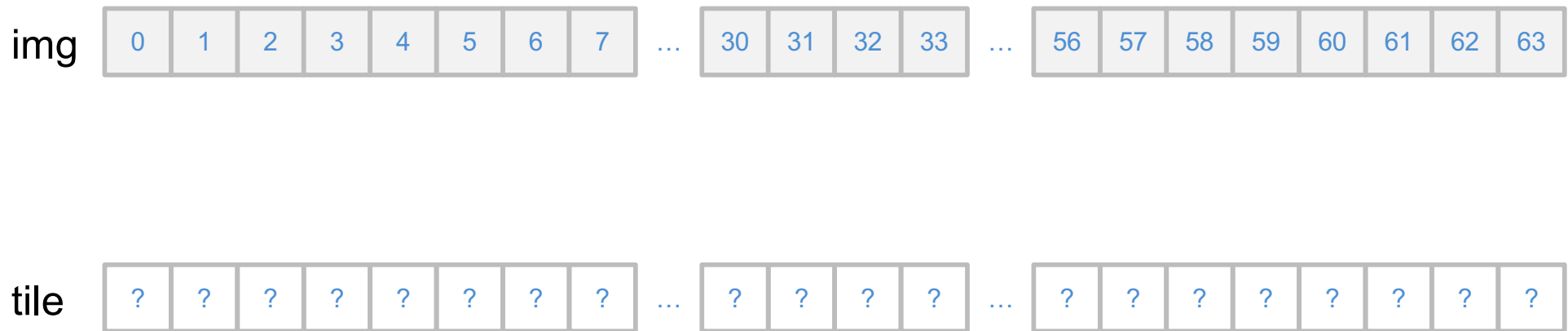
```
./run -g -k transpose -i 1000 -n -s <dim>
```



Back to “stripes”

Assuming TILE_W is a multiple of 64...

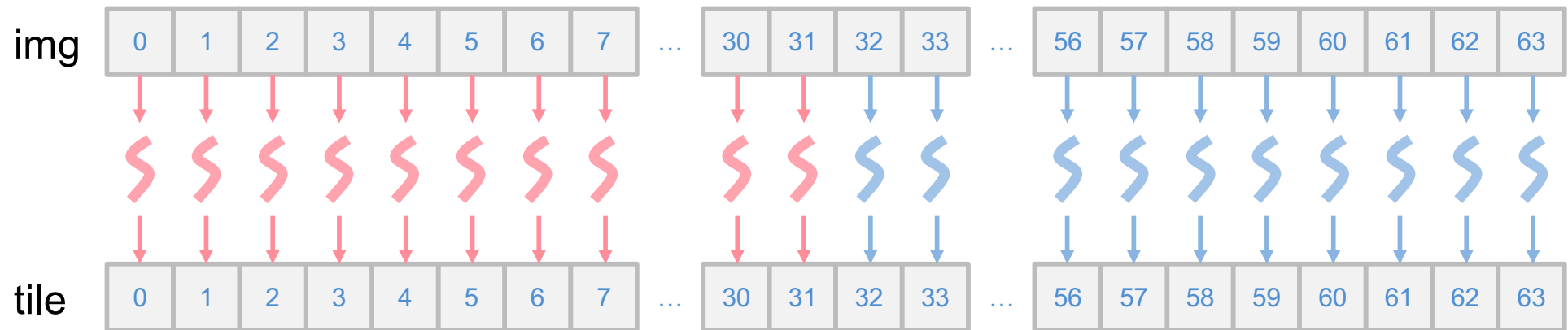
- Implementation of *1-pixel-width* stripes
 - Divergence avoiding
 - Memory coalescing



Back to “stripes”

Assuming TILE_W is a multiple of 64...

- Implementation of *1-pixel-width* stripes
 - Divergence avoiding
 - Memory coalescing



Back to “stripes”

Assuming TILE_W is a multiple of 64...

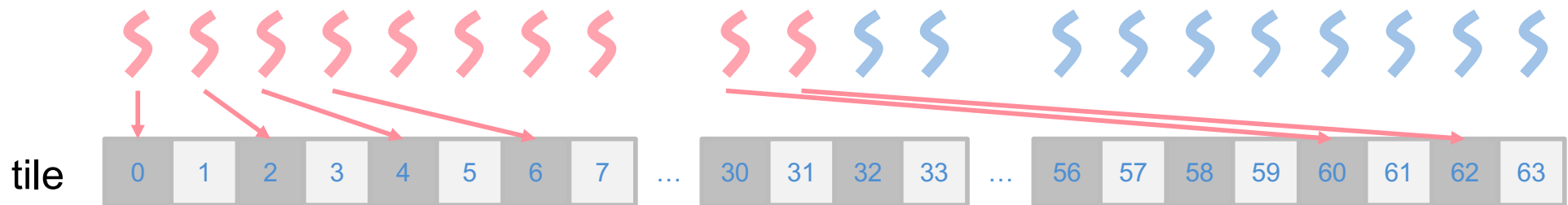
- Implementation of *1-pixel-width* stripes
 - Divergence avoiding
 - Memory coalescing



Back to “stripes”

Assuming TILE_W is a multiple of 64...

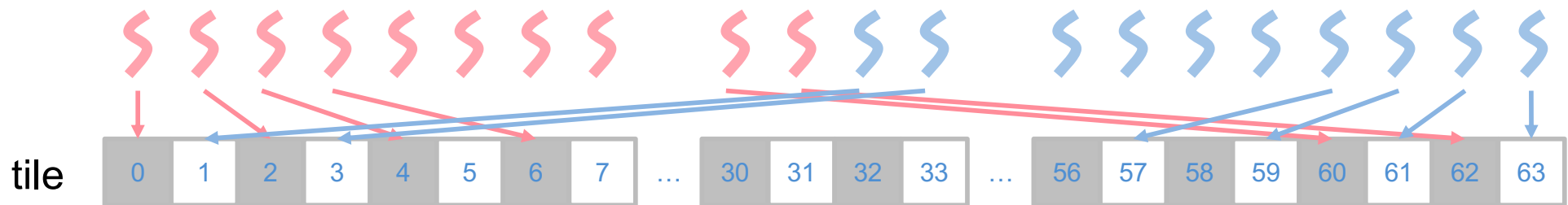
- Implementation of *1-pixel-width* stripes
 - Divergence avoiding
 - Memory coalescing



Back to “stripes”

Assuming TILE_W is a multiple of 64...

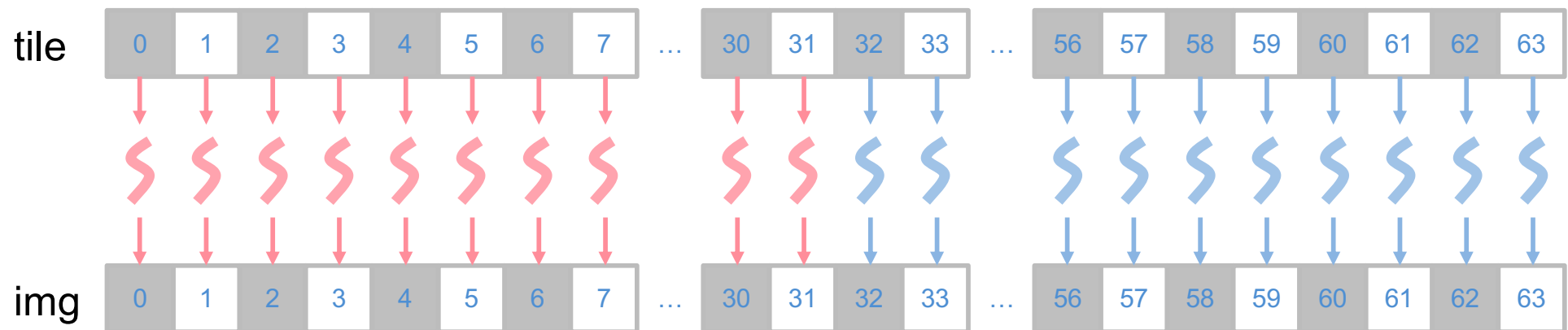
- Implementation of *1-pixel-width* stripes
 - Divergence avoiding
 - Memory coalescing



Back to “stripes”

Assuming TILE_W is a multiple of 64...

- Implementation of *1-pixel-width* stripes
 - Divergence avoiding
 - Memory coalescing



Back to “stripes”

Assuming TILE_W is a multiple of 64...

- Implementation of *1-pixel-width* stripes
 - Divergence avoiding
 - Memory coalescing

```
__local unsigned tile [TILE_H][TILE_W];
unsigned y = get_global_id (1), yloc = get_local_id (1);
unsigned x = get_global_id (0), xloc = get_local_id (0);
unsigned index = 2 * xloc;

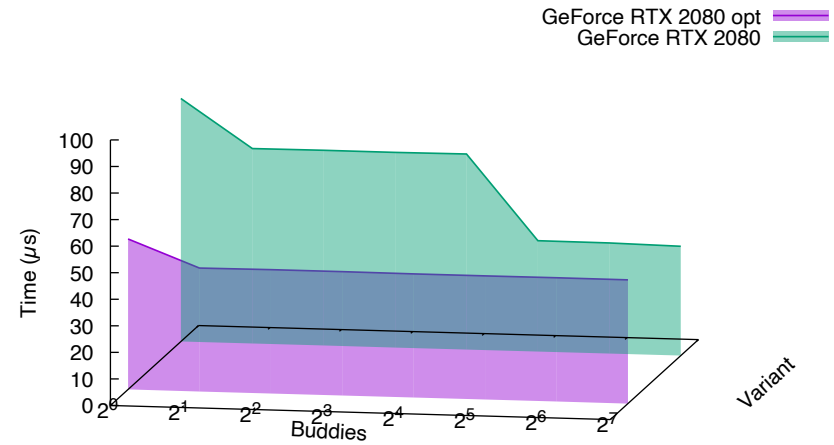
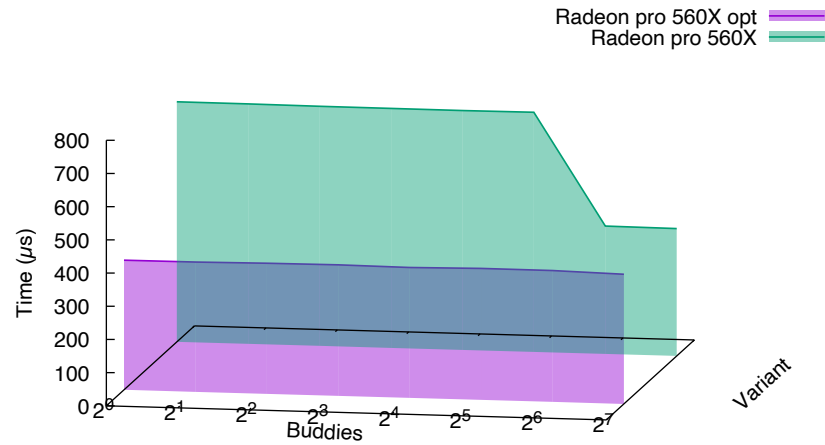
tile[yloc][xloc] = in [y * DIM + x];
barrier (CLK_LOCAL_MEM_FENCE);

if (index < get_local_size (0)) {
    tile [yloc][index] = darken (tile [yloc][index]);
} else {
    index += - get_local_size (0) + 1;
    tile [yloc][index] = brighten (tile [yloc][index]);
}

barrier (CLK_LOCAL_MEM_FENCE);
out [y * DIM + x] = tile [yloc][xloc];
```

Optimized version of stripes

```
./run -k stripes -g -tw 256 -th 1 -n ...
```

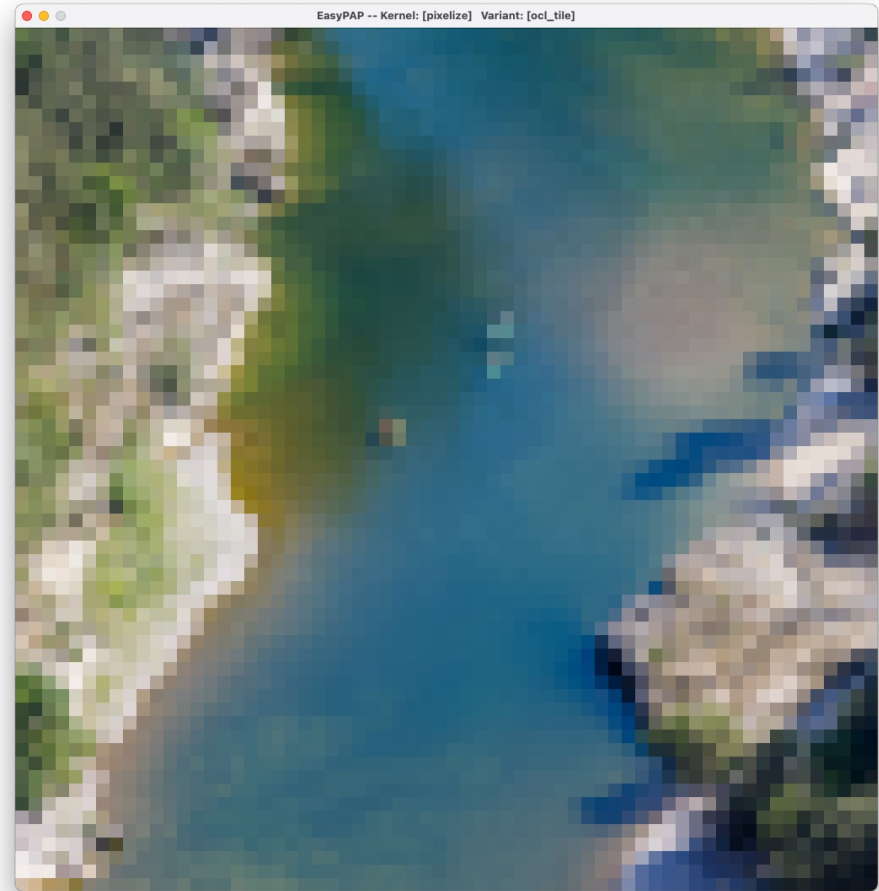


Back to pixelization

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



Simplified version

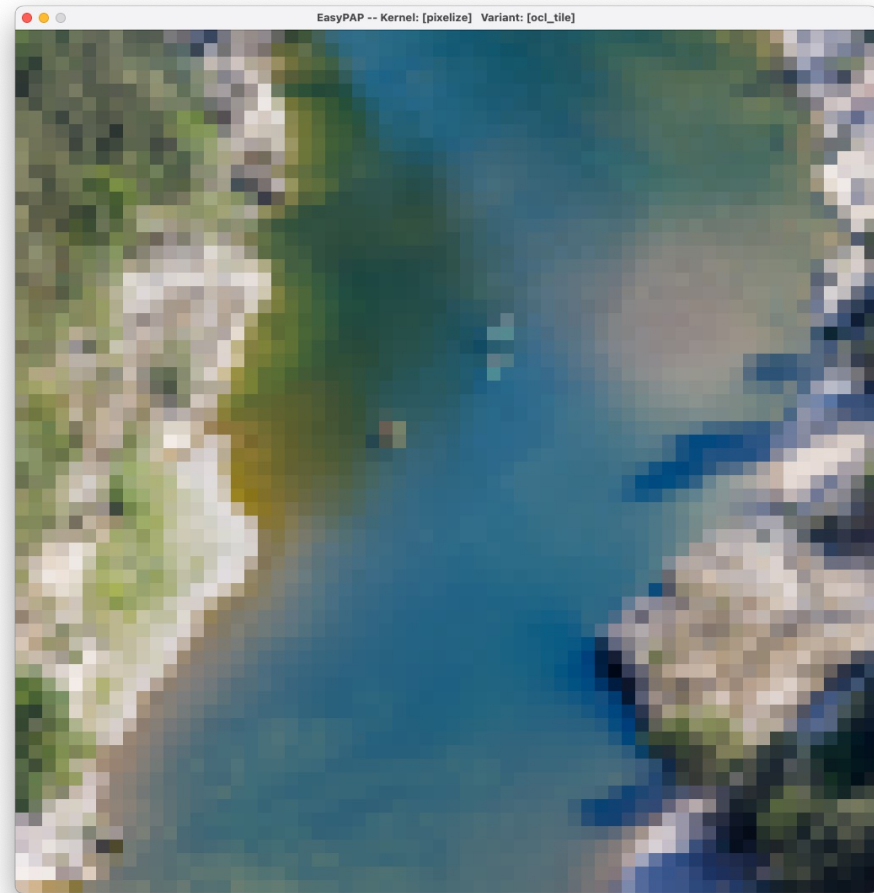


Expected version

Pixelization

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

- Pixelizing
 - All pixels within a tile adopt the same color
 - Average color of all pixels
- For each tile:
 - $\text{Sum} += \text{color of each pixel}$
 - $\text{Avg} = \text{Sum} / \#\text{pixels}$
 - All pixels take the Avg color
- First step is a 2D reduction



Expected version

Pixelization

Computing the sum of all pixels of a tile

- 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;`

```
__kernel void pixelize_ocl (  
    __global unsigned *in)  
{  
    __local int4 tile [TILE_H][TILE_W];  
    int x = get_global_id (0);  
    int y = get_global_id (1);  
    int xloc = get_local_id (0);  
    int yloc = get_local_id (1);  
  
    tile [yloc][xloc] =  
        color_to_int4 (in [y * DIM + x]);  
    ...  
}
```

Pixelization

Computing the sum of all pixels of a tile

- Reduction
 - We first cache pixels into local memory
 - Then we can perform our 2D reduction inside tile

```
__kernel void pixelize_ocl (
    __global unsigned *in)
{
    __local int4 tile [TILE_H][TILE_W];
    int x = get_global_id (0);
    int y = get_global_id (1);
    int xloc = get_local_id (0);
    int yloc = get_local_id (1);

    tile [yloc][xloc] =
        color_to_int4 (in [y * DIM + x]);
    barrier (CLK_LOCAL_MEM_FENCE);
    ...
}
```


Pixelization

Computing the sum of all pixels of a tile

- Reduction
 - There is one thread per cell
 - To maximize throughput of load operation
 - How do we compute the sum of all cells?

```
__kernel void pixelize_ocl (  
    __global unsigned *in)  
{  
    __local int4 tile [TILE_H][TILE_W];  
    int x = get_global_id (0);  
    int y = get_global_id (1);  
    int xloc = get_local_id (0);  
    int yloc = get_local_id (1);  
  
    tile [yloc][xloc] =  
        color_to_int4 (in [y * DIM + x]);  
    barrier (CLK_LOCAL_MEM_FENCE);  
    ...  
}
```

Pixelization

Computing the sum of all pixels of a tile

- Let's consider tiles of 8x4 cells
 - There is one thread per cell
 - To maximize throughput of load operation
 - How do we compute the sum of all cells?

1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1

Pixelization

Computing the sum of all pixels of a tile

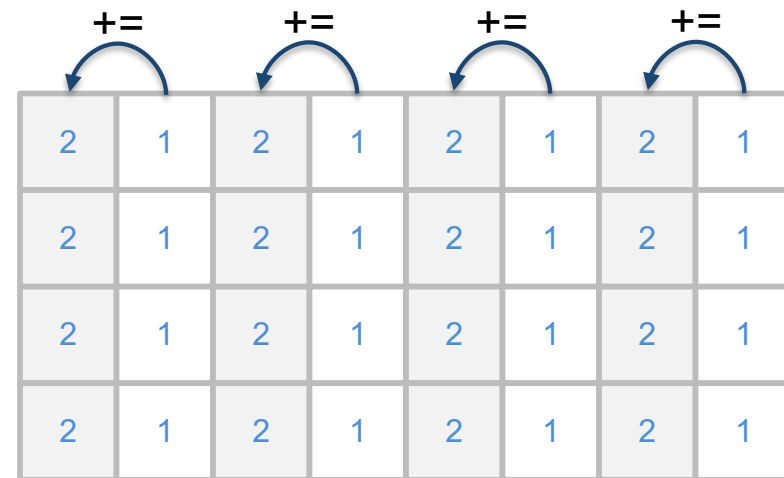
- Let's consider tiles of 8x4 cells
 - There is one thread per cell
 - To maximize throughput of load operation
 - How do we compute the sum of all cells?
 - Well, we could perform a first wave of 4x4 additions
 - 4 additions per row

1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1

Pixelization

Computing the sum of all pixels of a tile

- Let's consider tiles of 8x4 cells
 - There is one thread per cell
 - To maximize throughput of load operation
 - How do we compute the sum of all cells?
 - Well, we could perform a first wave of 4x4 additions
 - 4 additions per row

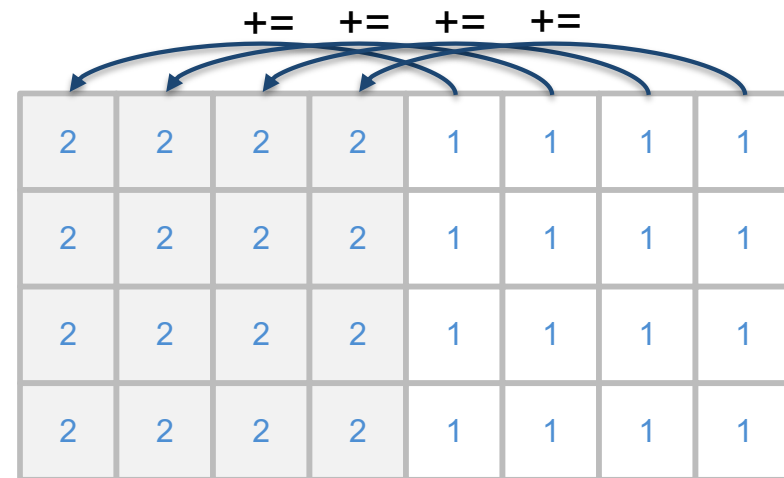


This way...

Pixelization

Computing the sum of all pixels of a tile

- Let's consider tiles of 8x4 cells
 - There is one thread per cell
 - To maximize throughput of load operation
 - How do we compute the sum of all cells?
 - Well, we could perform a first wave of 4x4 additions
 - 4 additions per row

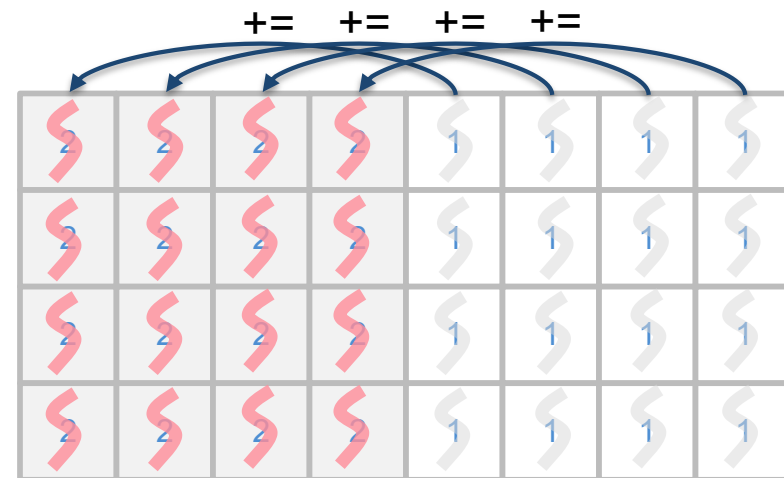


Or that way...

Pixelization

Computing the sum of all pixels of a tile

- Let's consider tiles of 8x4 cells
 - There is one thread per cell
 - To maximize throughput of load operation
 - How do we compute the sum of all cells?
 - Well, we could perform a first wave of 4x4 additions
 - 4 additions per row



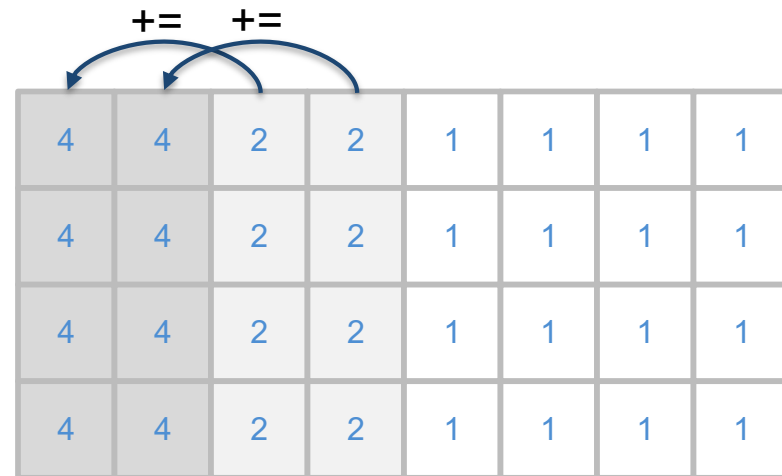
Half of threads do not work

```
if (xloc < 4)
    tile [yloc][xloc] += tile [yloc][xloc + 4];
```

Pixelization

Computing the sum of all pixels of a tile

- Next step
 - Second wave of 2x4 additions
2 additions per row



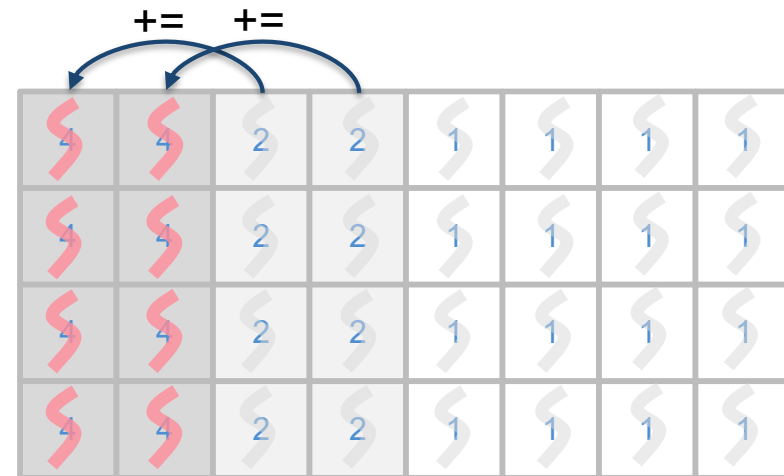
Pixelization

Computing the sum of all pixels of a tile

- Next step
 - Second wave of 2x4 additions

2 additions per row

Only $\frac{1}{4}$ of threads participate

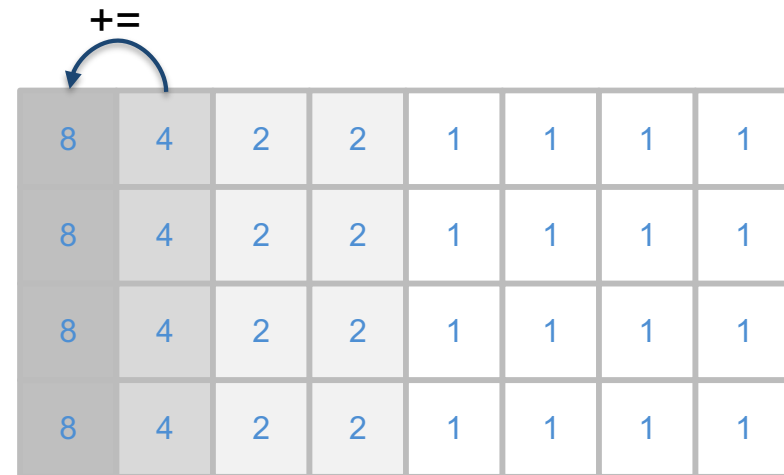


```
if (xloc < 2)
    tile [yloc][xloc] += tile [yloc][xloc + 2];
```


Pixelization

Computing the sum of all pixels of a tile

- Next step
 - Second wave of 2x4 additions
 - 2 additions per row
 - Only 1/8th of threads participate



```
if (xloc < 1)
    tile [yloc][xloc] += tile [yloc][xloc + 1];
```

Pixelization

Computing the sum of all pixels of a tile

- Now what?

8	4	2	2	1	1	1	1
8	4	2	2	1	1	1	1
8	4	2	2	1	1	1	1
8	4	2	2	1	1	1	1

Pixelization

Computing the sum of all pixels of a tile

- Now what?
 - We sum up the cells vertically, but only for the first column
 - Avoid wasting local memory bandwidth

8	4	2	2	1	1	1	1
8	4	2	2	1	1	1	1
8	4	2	2	1	1	1	1
8	4	2	2	1	1	1	1

Pixelization

Computing the sum of all pixels of a tile

- Now what?
 - We sum up the cells vertically, but only for the first column
 - Avoid wasting local memory bandwidth
 - Only two threads participate



```
if (xloc == 0) {  
    if (yloc < 2)  
        tile [yloc][0] += tile [yloc + 2][0];  
}
```

Pixelization

Computing the sum of all pixels of a tile

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

32	4	2	2	1	1	1	1
16	4	2	2	1	1	1	1
8	4	2	2	1	1	1	1
8	4	2	2	1	1	1	1

```
if (xloc == 0) {  
    if (yloc < 1)  
        tile [yloc][0] += tile [yloc + 1][0];  
}
```

Pixelization

Computing the sum of all pixels of a tile

```
{
  __local int4 tile [TILE_H][TILE_W];
  int x    = get_global_id (0);
  int y    = get_global_id (1);
  int xloc = get_local_id  (0);
  int yloc = get_local_id  (1);

  tile [yloc][xloc] = color_to_int4 (in [y * DIM + x]);

  // Averaging each line
  for (int d = TILE_W >> 1; d > 0; d >>= 1) {
    barrier (CLK_LOCAL_MEM_FENCE);
    if (xloc < d)
      tile [yloc][xloc] += tile [yloc][xloc + d];
  }

  // Averaging first column only
  for (int d = TILE_H >> 1; d > 0; d >>= 1) {
    barrier (CLK_LOCAL_MEM_FENCE);
    if (xloc == 0 && yloc < d)
      tile [yloc][xloc] += tile [yloc + d][xloc];
  }

  barrier (CLK_LOCAL_MEM_FENCE);
  in [y * DIM + x] = int4_to_color (tile [0][0] / (int4)(TILE_W * TILE_H));
}
```

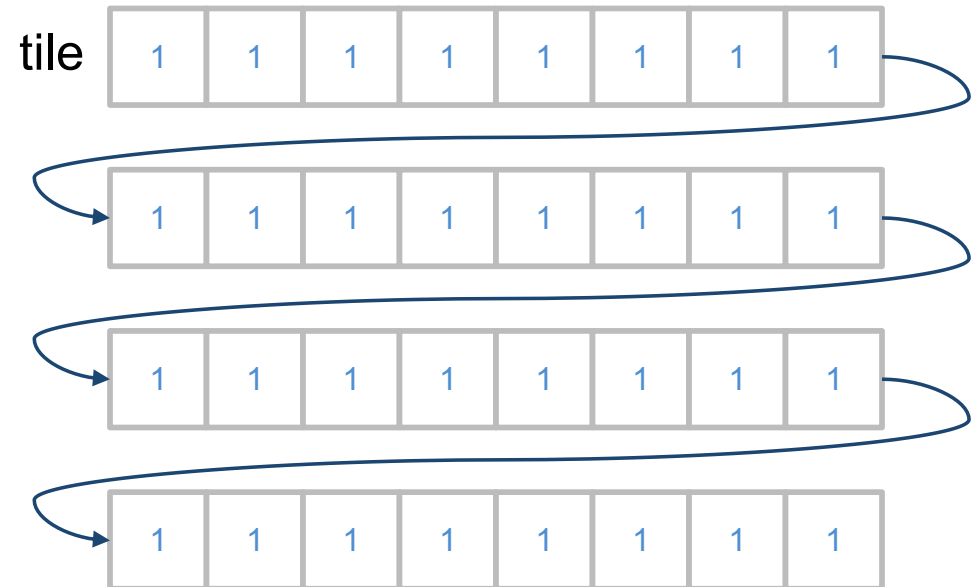
Pixelization

Computing the sum of all pixels of a tile

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

```
__local int4 tile[TILE_H * TILE_W];  
  
int loc = get_local_id (1) * TILE_W +  
         get_local_id (0);
```

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



Pixelization

Computing the sum of all pixels of a tile

```
{
  __local int4 tile [TILE_H * TILE_W];
  int x    = get_global_id (0);
  int y    = get_global_id (1);
  int loc  = get_local_id (1) * TILE_W + get_local_id (0);

  tile [loc] = color_to_int4 (in [y * DIM + x]);

  for (int d = (TILE_W * TILE_H) >> 1; d > 0; d >>= 1) {

    barrier (CLK_LOCAL_MEM_FENCE);

    if (loc < d)
      tile [loc] += tile [loc + d];
  }

  barrier (CLK_LOCAL_MEM_FENCE);

  in [y * DIM + x] = int4_to_color (tile [0] / (int4) (TILE_W * TILE_H));
}
```


Pixelization

Who on earth loves pixelization?



Final notes about reductions

- What if TILE size exceeds workgroup maximum size?
 - We can no longer use our method...
- See `pixelize_ocl_big`

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

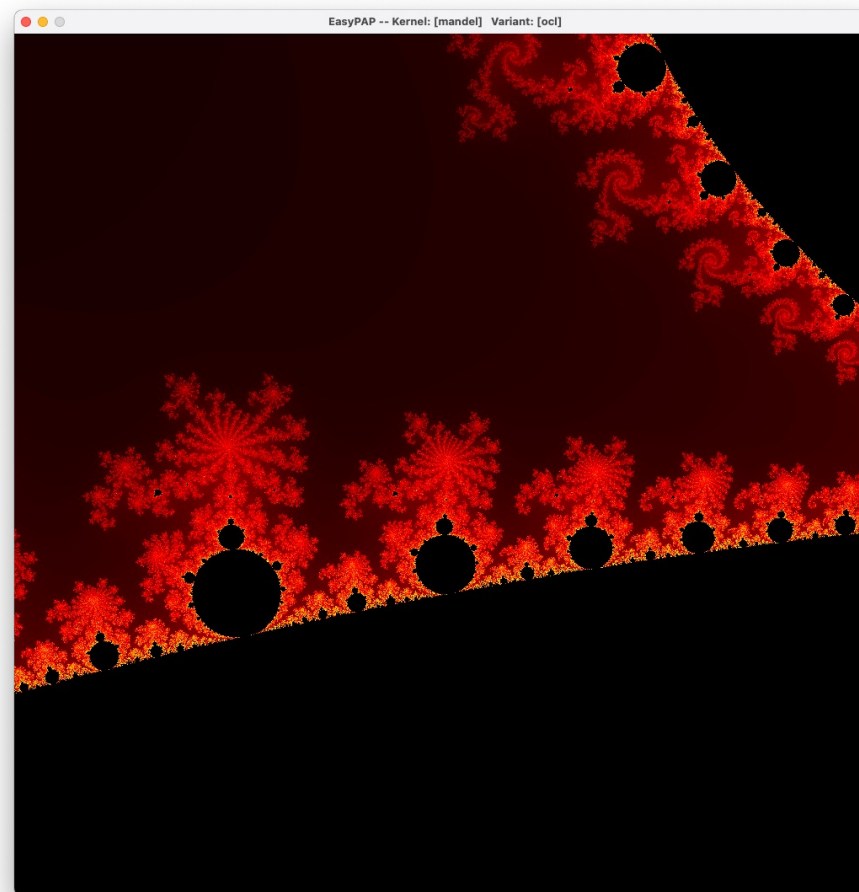
Final notes about reductions

- Detecting kernel stability on GPU is “kind of a reduction”
 - Each thread has a 'changed' value
 - We want to compute a unique 'stable' value
 - Is it worth doing it entirely on the GPU?

Some OpenCL kernels are straightforward...

Mandelbrot

- Mandelbrot is a compute-bound kernel
 - No memory access challenge
 - No data reuse
 - There is intra-warp divergence
 - As in our AVX version...



Some OpenCL kernels are straightforward...

Mandelbrot

- Mandelbrot is a compute-bound kernel
 - No memory access challenge
 - No data reuse
 - There is intra-warp divergence
 - As in our AVX version...
 - Code is similar to the sequential one!

```
__kernel void mandel_ocl (__global unsigned *img,
                          float leftX, float xstep,
                          float topY, float ystep,
                          unsigned MAX_ITERATIONS)
{
    int i = get_global_id (1);
    int j = get_global_id (0);
    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;
        float 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) ?
        mandel_iter2color (iter) :
        0x000000FF; // black
}
```

Mixing OpenCL and OpenMP

Hybrid Computing

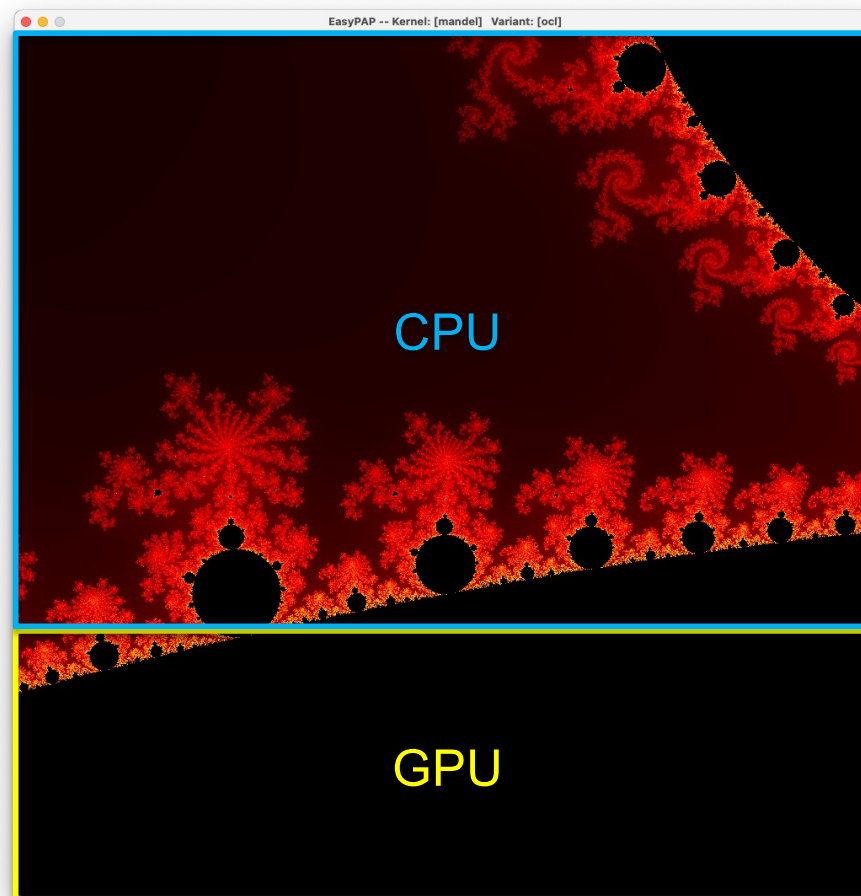
- Implementing a CPU+GPU

Mandelbrot should be a no-brainer

- No data exchange needed between iterations!
 - Things would be different for a stencil code
 - How about a hybrid abelian sandpile uh? 😊

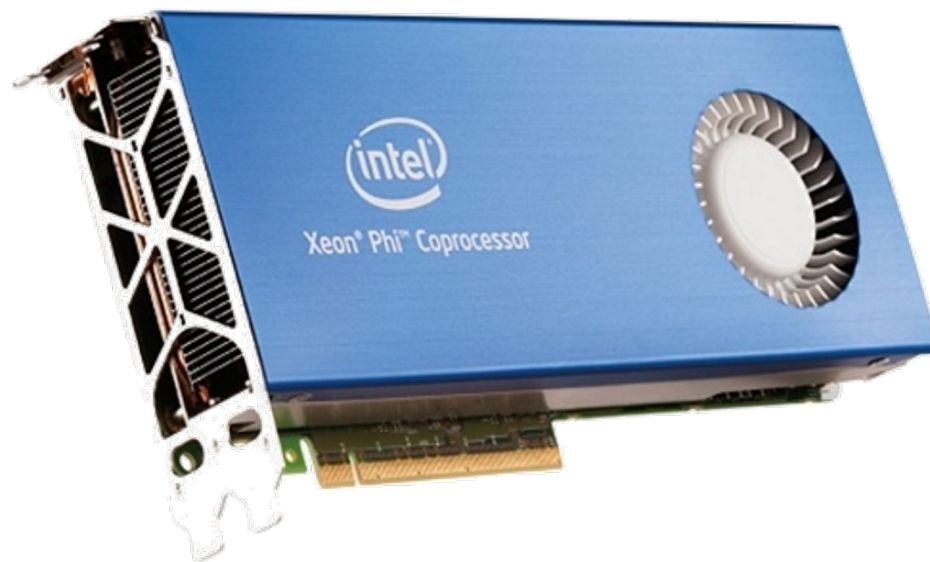
- Fixed partitioning
 - CPU takes n tile rows
 - GPU takes $\text{NB_TILES_Y} - n$ tile rows

Go and try with EasyPAP!



OpenCL on regular CPU architectures

- Intel Xeon Phi coprocessor
 - [KNC, 2012]
 - 61 cores, in-order, superscalar (1,1 - 1,3GHz, 22nm)
 - 4-way hyperthreading (244 threads)
 - 8 – 16 GB GDDR5
 - 5,5GT/s, 512 bits
 - Cache
 - L1 32KB/core
 - L2 512KB/core
 - ~1Tflop
 - 225-300W
- Xeon Phi is
 - A PCIe accelerator board
 - A full x86 PC running Linux



OpenCL on Xeon Phi

Implicit Vectorization

- The OpenCL runtime system spawns 240 OS threads
 - OS threads pinned on each core
- OpenCL workgroups are dispatched among threads
 - Each workgroup is executed sequentially by one thread
 - At least 240 workgroups are needed to feed all cores
- Kernels are implicitly vectorized along dimension 0
 - Work items are grouped to form `get_local_size(0)/16` vectors

```
__Kernel void foo(...)  
  For (int i = 0; i < get_local_size(2); i++)  
    For (int j = 0; j < get_local_size(1); j++)  
      For (int k = 0; k < get_local_size(0); k += VECTOR_SIZE)  
        Vectorized_Kernel_Body;
```

OpenCL on Xeon Phi

Code divergence within workgroups

- Conditional code is not harmful when all work items (within a WG) are guaranteed to execute the same branch

```
- if (get_local_id (1) == y)
    foo();
```

- In other cases, code has to be “masked” and both IF & ELSE parts are executed for all work items

```
if (get_global_id (0) % n == 0)
    res = IF_code ();
else
    res = ELSE_code ();
```

```
gid16 = get16_global_id (0);
Mask = compare16int ((gid % broadcast16 (32)), 0)
Res_if = IF_code ();
Res_else = ELSE_code ();
Res = (res_if & mask) | (res_else & not(mask));
```

OpenCL and Beyond

- Many topics we did not cover
 - Unified Memory
 - Atomic operations
 - Out-of-order queues, synchronizations through events
 - Etc.
- OpenCL 3.0 (December 2020)
 - Most features of OpenCL 2.x are... optional

OpenCL vs OpenMP on CPUs: the battle for SIMD

- OpenMP is a higher-level, far more versatile approach
 - Support for irregular, task parallelism
 - Incremental parallelization of the code
- OpenCL is a low-level, accelerator-focused approach
 - "Think Massively Parallel"
 - Hundreds of thousands of thread execute the code right from the start
 - Forget about global synchronizations, collective data movements
 - Too expensive
- On massively multicore machines
 - Global synchronizations/data exchanges will also become too expensive
 - Do we really want to schedule individual tasks on thousands of cores?
 - Starting from a sequential program is not always a good idea
 - Maybe the truth is somewhere in between 😊

OpenMP for accelerators

OpenMP 4.0

- Since OpenMP 4.0 specification

```
#pragma omp target device(0) map(to:A, B) map(tofrom:C)
#pragma omp teams num_teams(num_blocks) num_threads(bsize)
#pragma omp distribute
for (int i = 0; i < N; i ++)
```

```
    #pragma omp parallel for
    for (int j = 0; j < N; j++)
        for (int k = 0; k < N; k++)
            C[i][j] += A[i][k] * B[k][j];
```

OpenMP for accelerators

OpenMP 4.0

- OpenMP 4.0 (2013)
 - We still miss full-featured compilers
 - IBM XL compiler
 - Intel icx
- OpenACC (2012)
 - Cray, CAPS, Nvidia and PGI
 - PG Compiler
 - Efficient code generation on Nvidia GPUs
 - Intended to serve as a temporary solution
 - Until OpenMP integrates

Prepare Yourself for Total Meltdown!



OPEN SEASON WILL RETURN