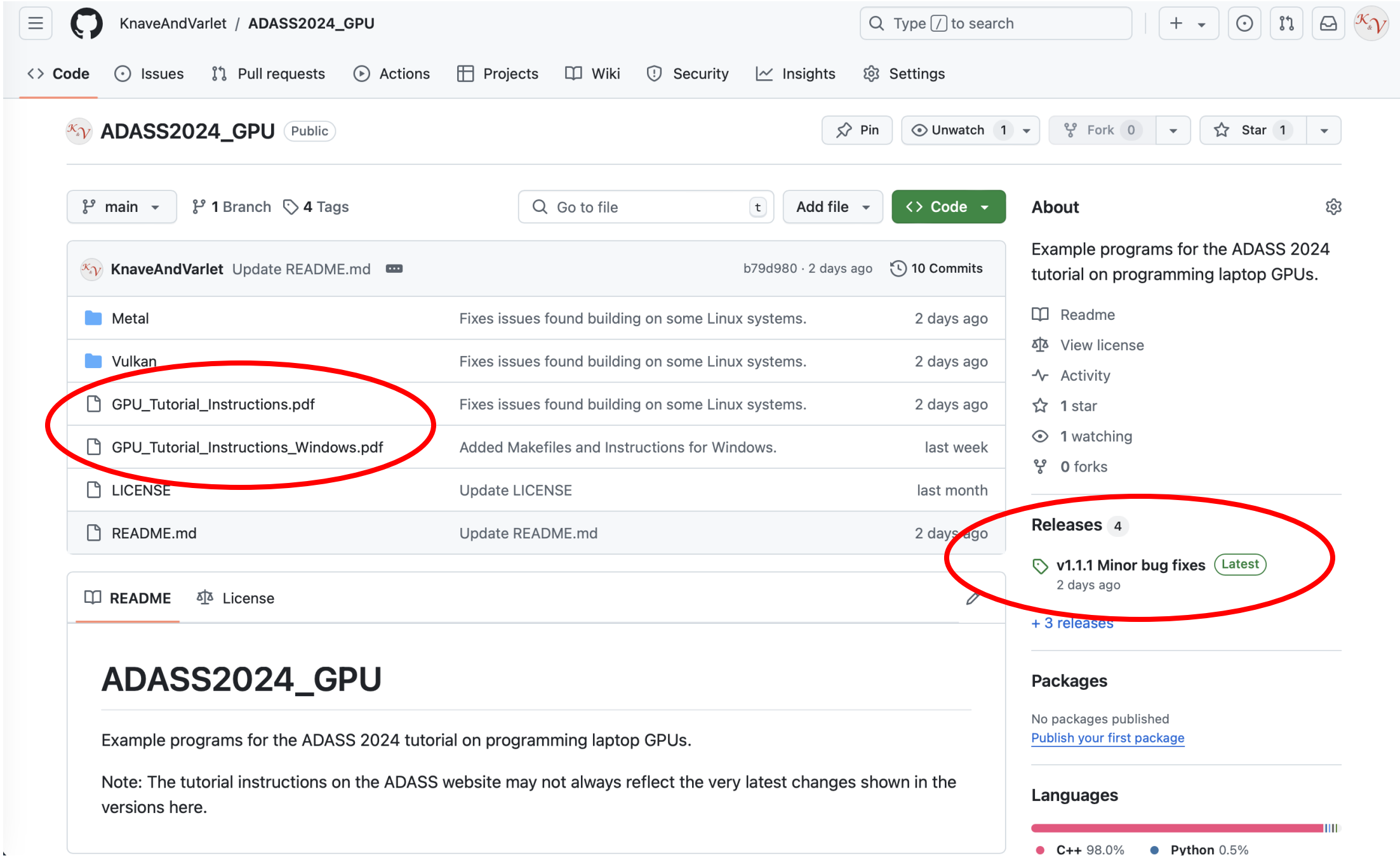
Github.com/KnaveAndVarlet/ADASS2024_GPU


KnaveAndVarlet / ADASS2024_GPU
Type / to search
Code
Issues
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
ADASS2024_GPU
Public
Pin
Unwatch 1
Fork 0
Star 1
main
1 Branch
4 Tags
Go to file
t
Add file
Code
KnaveAndVarlet Update README.md
b79d980 · 2 days ago
10 Commits
Metal
Fixes issues found building on some Linux systems.
2 days ago
Vulkan
Fixes issues found building on some Linux systems.
2 days ago
GPU_Tutorial_Instructions.pdf
Fixes issues found building on some Linux systems.
2 days ago
GPU_Tutorial_Instructions_Windows.pdf
Added Makefiles and Instructions for Windows.
last week
LICENSE
Update LICENSE
last month
README.md
Update README.md
2 days ago
README
License
ADASS2024_GPU
Example programs for the ADASS 2024 tutorial on programming laptop GPUs.
Note: The tutorial instructions on the ADASS website may not always reflect the very latest changes shown in the versions here.
About
Example programs for the ADASS 2024 tutorial on programming laptop GPUs.
Readme
View license
Activity
1 star
1 watching
0 forks
Releases 4
v1.1.1 Minor bug fixes
Latest
2 days ago
+ 3 releases
Packages
No packages published
Publish your first package
Languages
C++ 98.0%
Python 0.5%

sudo apt update
(sudo apt install build-essential)
sudo apt install vulkan-tools
sudo apt install libvulkan-dev
sudo apt install vulkan-validationlayers-dev spirv-tools
sudo apt install libglfw3-dev

Ubuntu

Minimal Vulkan and GLFW installation
(and C++ tools)

sudo dnf check-update
sudo dnf upgrade
(sudo dnf install gcc-c++)
(sudo dnf install make)
sudo dnf install vulkan-tools
sudo dnf install vulkan-loader-devel
sudo dnf install mesa-vulkan-devel vulkan-validation-layers-devel
sudo dnf install glfw-devel

Fedora

```
float gridXcent = gridSize.x * 0.5;
float gridYcent = gridSize.y * 0.5;
float x0 = args->xCent + (index2.x - gridXcent) * args->dX;
float y0 = args->yCent + (index2.y - gridYcent) * args->dY;

float x = 0.0;
float y = 0.0;
uint iteration = 0;
uint max_iteration = args->iter;
float xtmp = 0.0;
while(((x * x) + (y * y) <= 4.0) && (iteration < max_iteration))
{
    xtmp = (x + y) * (x - y) + x0;
    y = (2.0 * x * y) + y0;
    x = xtmp;
    iteration += 1;
}

float value = iteration;
if (iteration >= max_iteration) value = 0.0;
uint index = index2.y * gridSize.x + index2.x;
out[index] = value;
```

Metal code

```
float gridXcent = args.nx * 0.5;
float gridYcent = args.ny * 0.5;
float x = args.xCent + (float(gl_GlobalInvocationID.x) - gridXcent) * args.dX;
float y = args.yCent + (float(gl_GlobalInvocationID.y) - gridYcent) * args.dY;

vec2 c = vec2(x,y);
vec2 z = vec2(0.0,0.0);
float n = 0.0;
const int M = args.iter;
for (int i = 0; i<M; i++)
{
  n++;
  z = vec2(z.x*z.x - z.y*z.y, 2.*z.x*z.y) + c;
  if (dot(z, z) > 4.0) break;
}
if (n >= M) n = 0.0;

imageData[args.nx * gl_GlobalInvocationID.y + gl_GlobalInvocationID.x] = n;
```

GLSL code

```
float InArray[NY][NX];
float OutArray[NY][NX];

for (int Iy = 0; Iy < NY; Iy++) {
    for (int Ix = 0; Ix < NX; Ix++) {
        OutArray[Iy][Ix] = InArray[Iy][Ix] + Ix + Iy;
    }
}
```

```
float InArray[NY][NX];
float OutArray[NY][NX];

for (int Iy = 0; Iy < NY; Iy++) {
    for (int Ix = 0; Ix < NX; Ix++) {
        OutArray[Iy][Ix] = InArray[Iy][Ix] + Ix + Iy;
    }
}
```

```
float* In = new float[Nx * Ny];
float* Out = new float[Nx * Ny];

for (int Iy = 0; Iy < Ny; Iy++) {
    for (int Ix = 0; Ix < Nx; Ix++) {
        Out[Iy * Nx + Ix] = In[Iy * Nx + Ix] + Ix + Iy;
    }
}
```

```
kernel void adder(device float *in [[ buffer(1) ]],
                  device float *out[[ buffer(2) ]],
                  uint2 index2 [[thread_position_in_grid]],
                  uint2 gridSize [[threads_per_grid]]) {

    uint nx = gridSize.x;
    uint ix = index2.x;
    uint iy = index2.y;
    out[iy * nx + ix] = in[iy * nx + ix] + iy + ix;
}
```

Metal Adder GPU code

```
struct AdderArgs {int nx; int ny; };
layout(std140,binding = 0) uniform paramBuf { AdderArgs args; };
layout(binding = 1) readonly buffer inBuf { float inputData[]; };
layout(binding = 2) writeonly buffer outBuf { float outputData[]; };

void main() {

    uint ix = gl_GlobalInvocationID.x;
    uint iy = gl_GlobalInvocationID.y;
    uint nx = args.nx;
    uint ny = args.ny;
    if (ix >= nx || iy >= ny) return;
    outputData[iy * nx + ix] = inputData[iy * nx + ix] + ix + iy;
}
```

Vulkan Adder GPU code

GRAPHICS REINVENTED

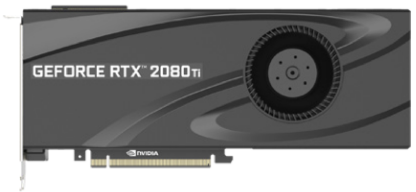
NVIDIA's newest flagship graphics card is a revolution in gaming realism and performance. Its powerful NVIDIA Turing™ GPU architecture, breakthrough technologies, and 11 GB of next-gen, ultra-fast GDDR6 memory make it the world's ultimate gaming GPU.

GeForce RTX™ graphics cards are powered by the Turing GPU architecture and the all-new RTX platform. This gives you up to 6× the performance of previous-generation graphics cards and brings the power of real-time ray tracing and AI to games.

When it comes to next-gen gaming, it's all about realism. GeForce RTX 2080 Ti is light years ahead of other cards, delivering truly unique real-time ray-tracing technologies for cutting-edge, hyper-realistic graphics.

KEY FEATURES

- Real-Time Ray Tracing
- NVIDIA® GeForce Experience™
- NVIDIA® Ansel
- NVIDIA® Highlights
- NVIDIA® G-SYNC™ Compatible



GeForce RTX™ 2080 Ti 11GB

SPECIFICATIONS

NVIDIA GeForce RTX™ 2080 Ti 11GB Blower	
PNY Part Number	**VCG2080T11BLMPB**
UPC Code	**751492621722**
Card Dimensions	**1.38" × 10.50" × 4.50", Dual-Slot**
Box Dimensions	**6.76" × 12.21" × 3.53"**
NVIDIA® CUDA Cores	**4352**
Clock Speed	**1350 MHz**
Boost Speed	**1545 MHz**
Memory Speed (Gbps)	**14**
Memory Size	**11GB GDDR6**
Memory Interface	**352-bit**
Memory Bandwidth (GB/sec)	**616**
TDP	**250 W**[5]
NVLink	**2-way**
Outputs	**DisplayPort 1.4 (x3), HDMI 2.0b, USB Type-C**
Multi-Screen	**Yes**
Resolution	**7680 × 4320 @60Hz (Digital)**[1]
Power Input	**Two 8-Pin**
Bus Type	**PCI-Express 3.0 ×16**

Chip

Apple M2 chip

8-core CPU with four performance cores and four efficiency cores

10-core GPU

16-core Neural Engine

100GB/s memory bandwidth

Media engine

Hardware-accelerated H.264, HEVC, ProRes and ProRes RAW

Video decode engine

Video encode engine

ProRes encode and decode engine

Apple M2 chip

8-core CPU with four performance cores and four efficiency cores

10-core GPU

16-core Neural Engine

100GB/s memory bandwidth

Media engine

Hardware-accelerated H.264, HEVC, ProRes and ProRes RAW

Video decode engine

Video encode engine

ProRes encode and decode engine

Apple M2 Pro chip

10-core CPU with six performance cores and four efficiency cores

16-core GPU

16-core Neural Engine

200GB/s memory bandwidth

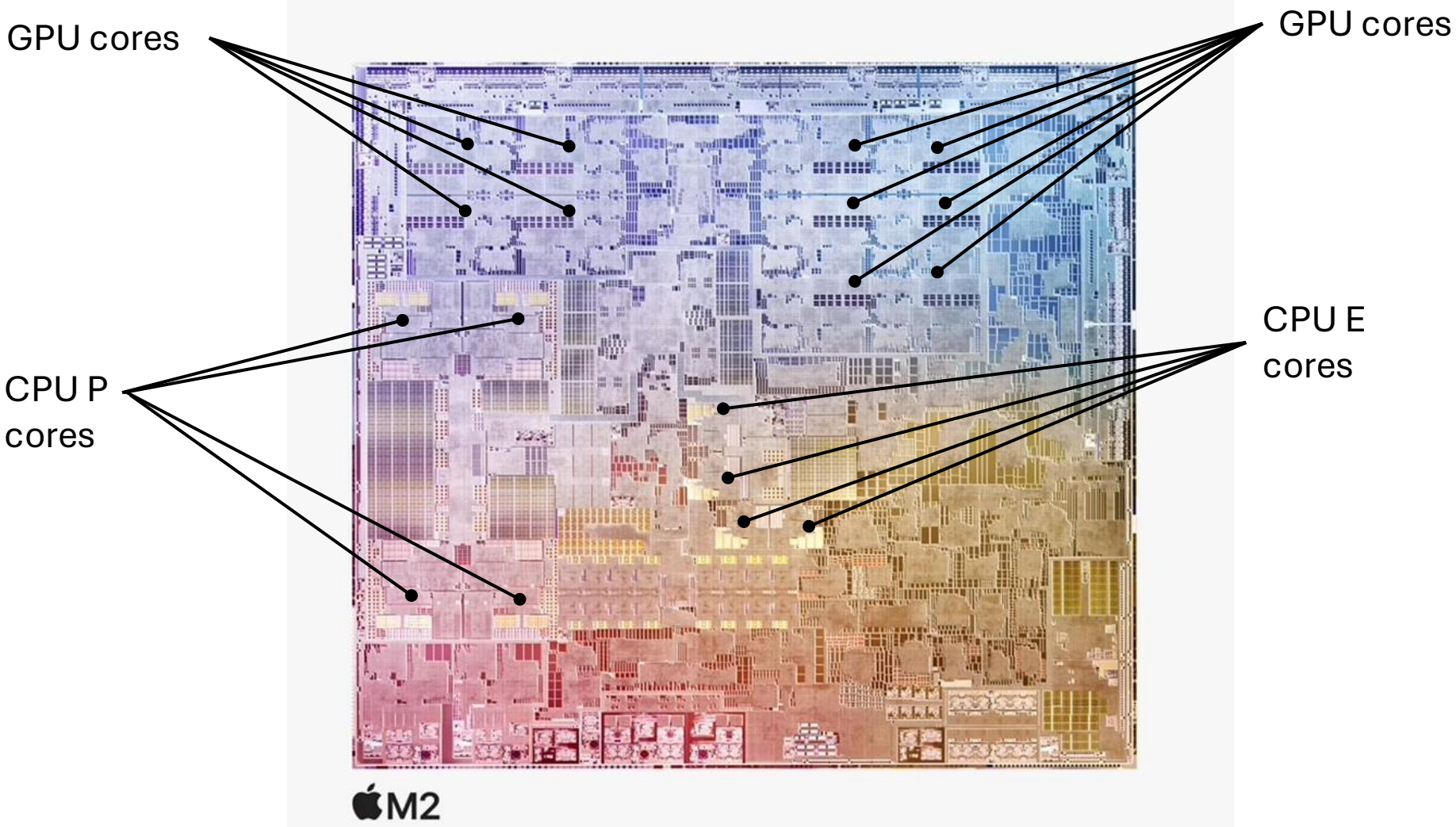
Media engine

Hardware-accelerated H.264, HEVC, ProRes and ProRes RAW

Video decode engine

Video encode engine

ProRes encode and decode engine



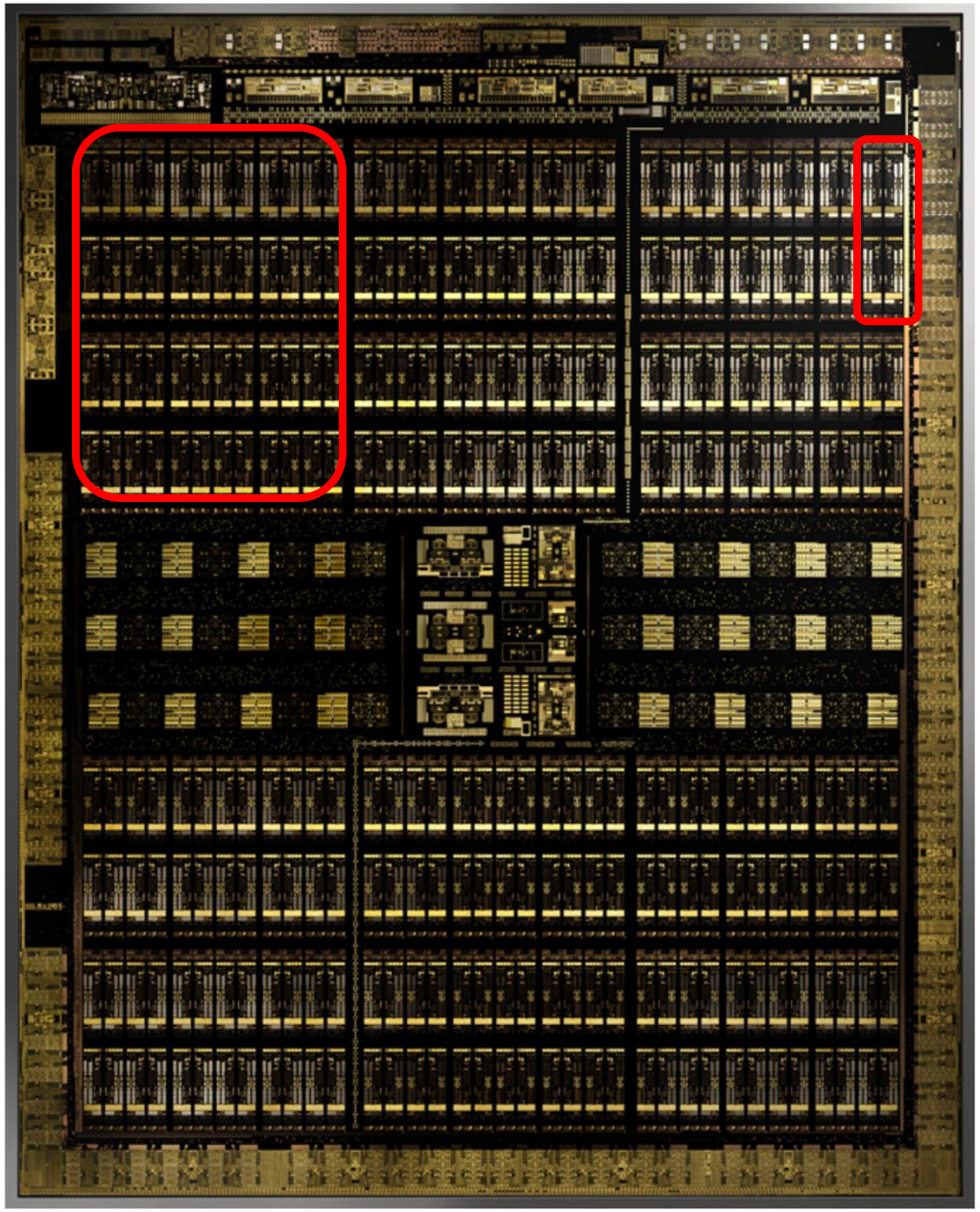
Apple Silicon M2

https://www.apple.com/au/newsroom/2022/06/apple-unveils-m2-with-breakthrough-performance-and-capabilities/

GPC – GPU Processing Cluster, with 12 SMs (Streaming Multiprocessors) each.

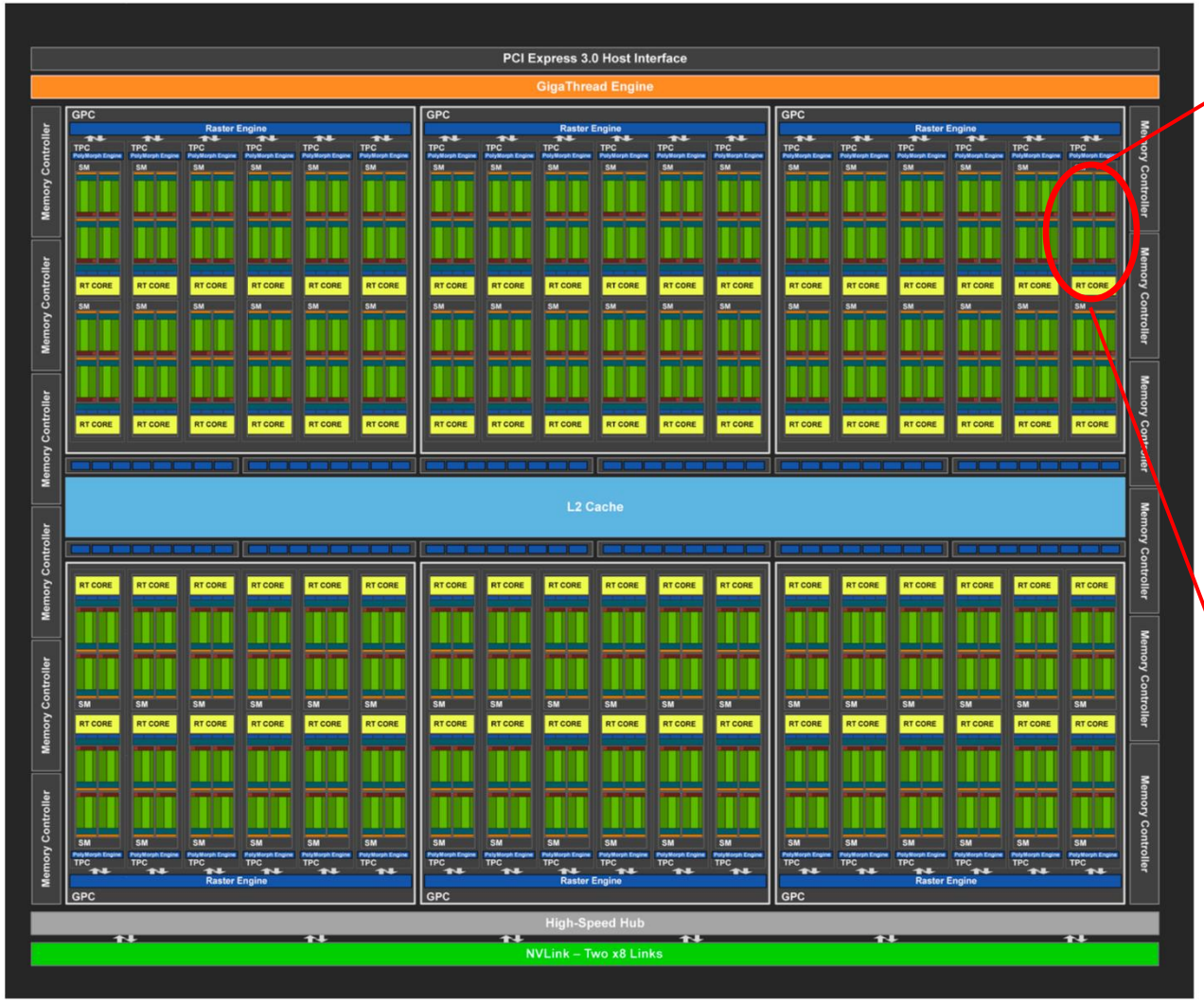
6 GPCs per device.

Total of 72 SMs per device.



Streaming multiprocessor

NVIDIA Turing TU102 GPU



Note: The TU102 GPU also features 144 FP64 units (two per SM), which are not depicted in this diagram. The FP64 TFLOP rate is 1/32nd the TFLOP rate of FP32 operations. The small number of FP64 hardware units are included to ensure any programs with FP64 code operates correctly.

Figure 2. Turing TU102 Full GPU with 72 SM Units

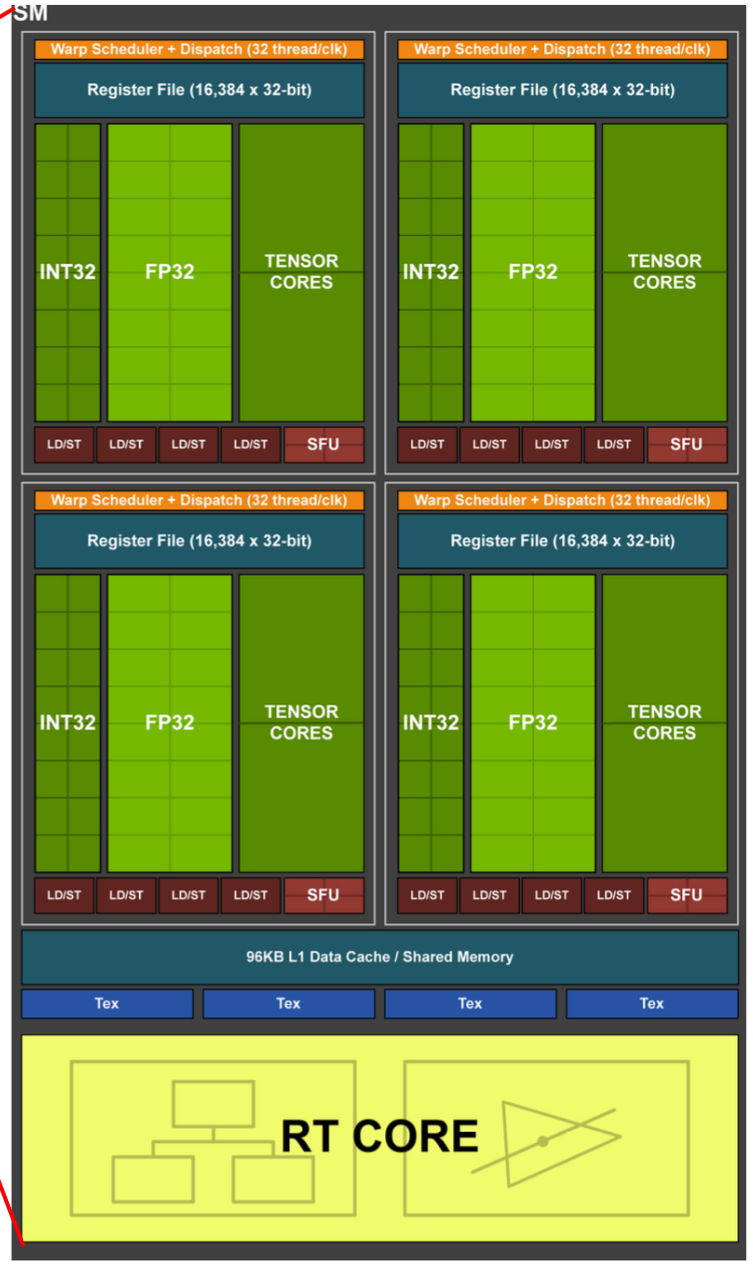
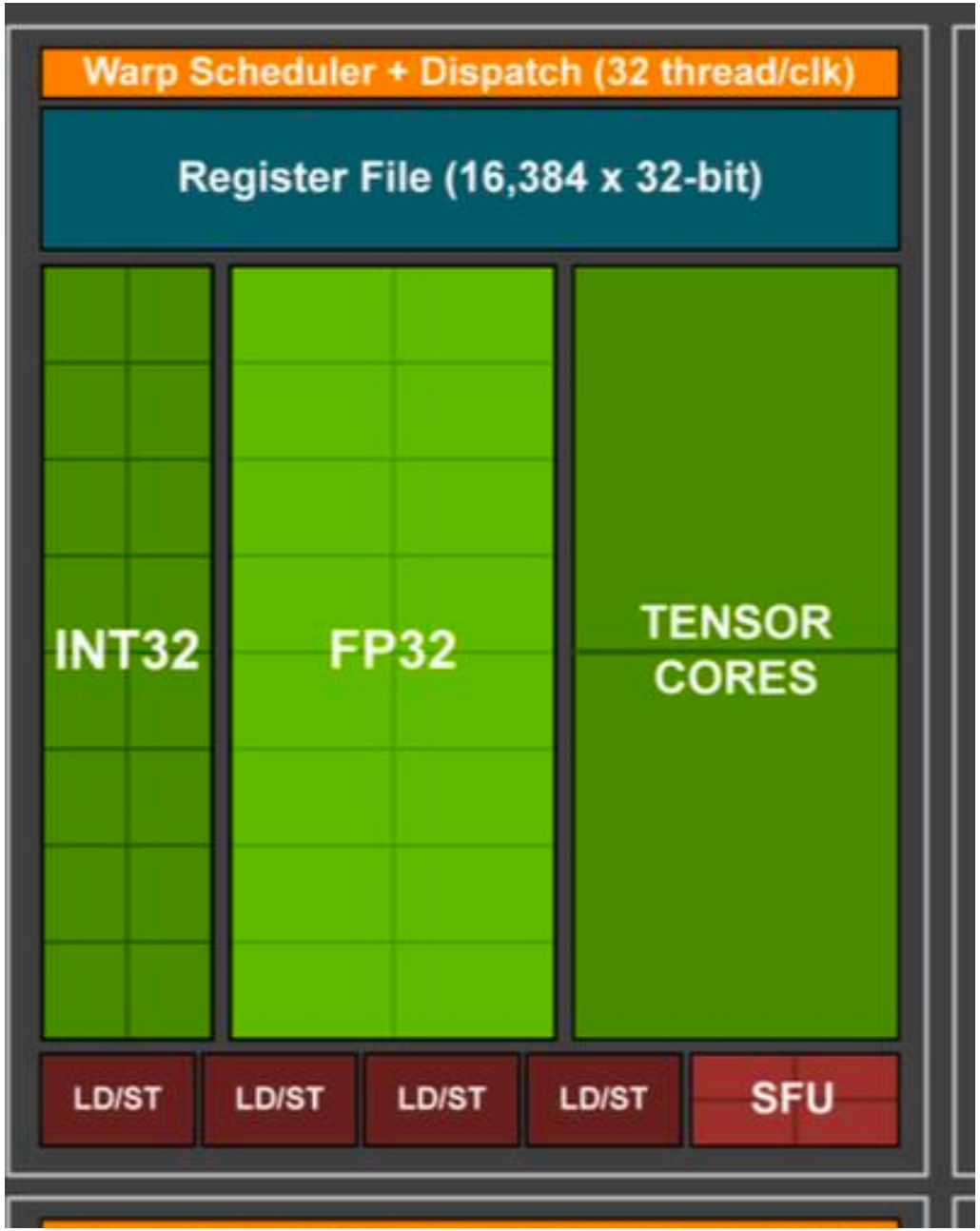


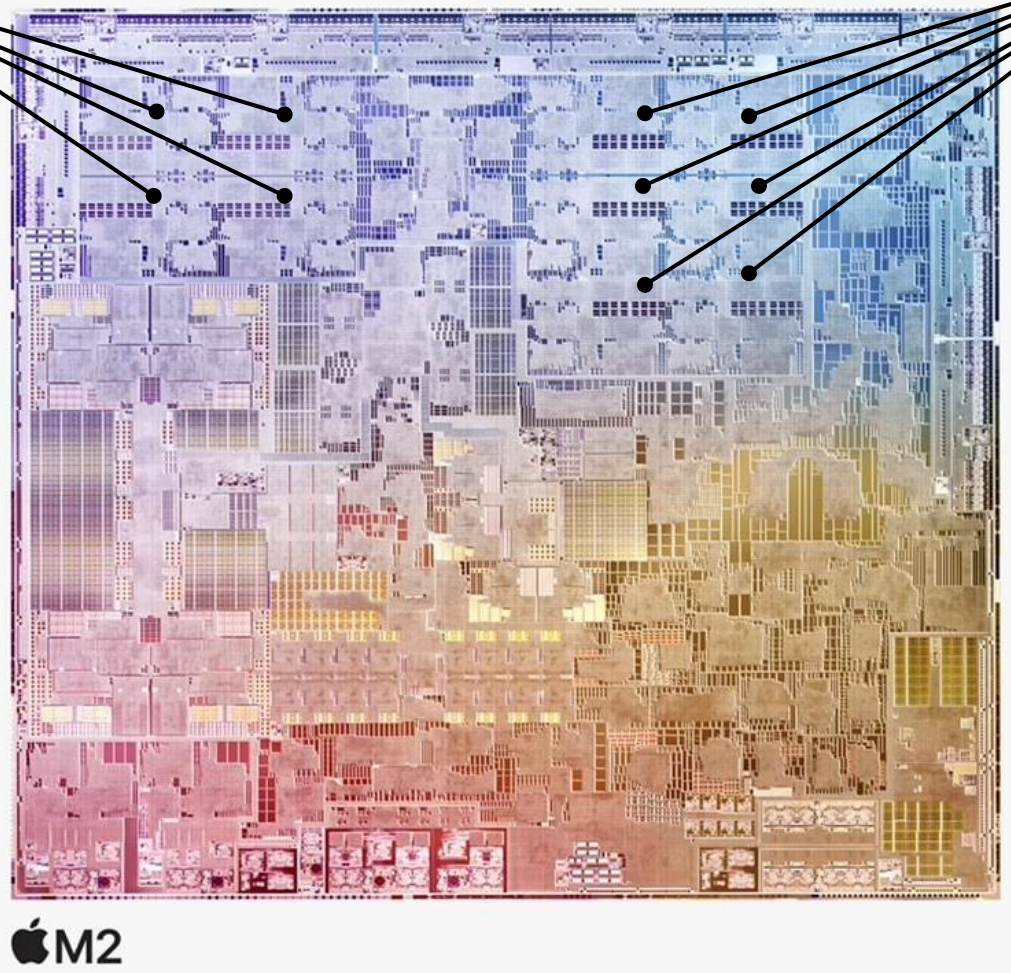
Figure 4. Turing TU102/TU104/TU106 Streaming Multiprocessor (SM)

Nvidia TU102
Streaming multiprocessor (SM)
sub-partition (SMSP)



GPU cores

GPU cores



Each of the 10 GPU core has
16 ‘execution units’ each with 8 ALUs.

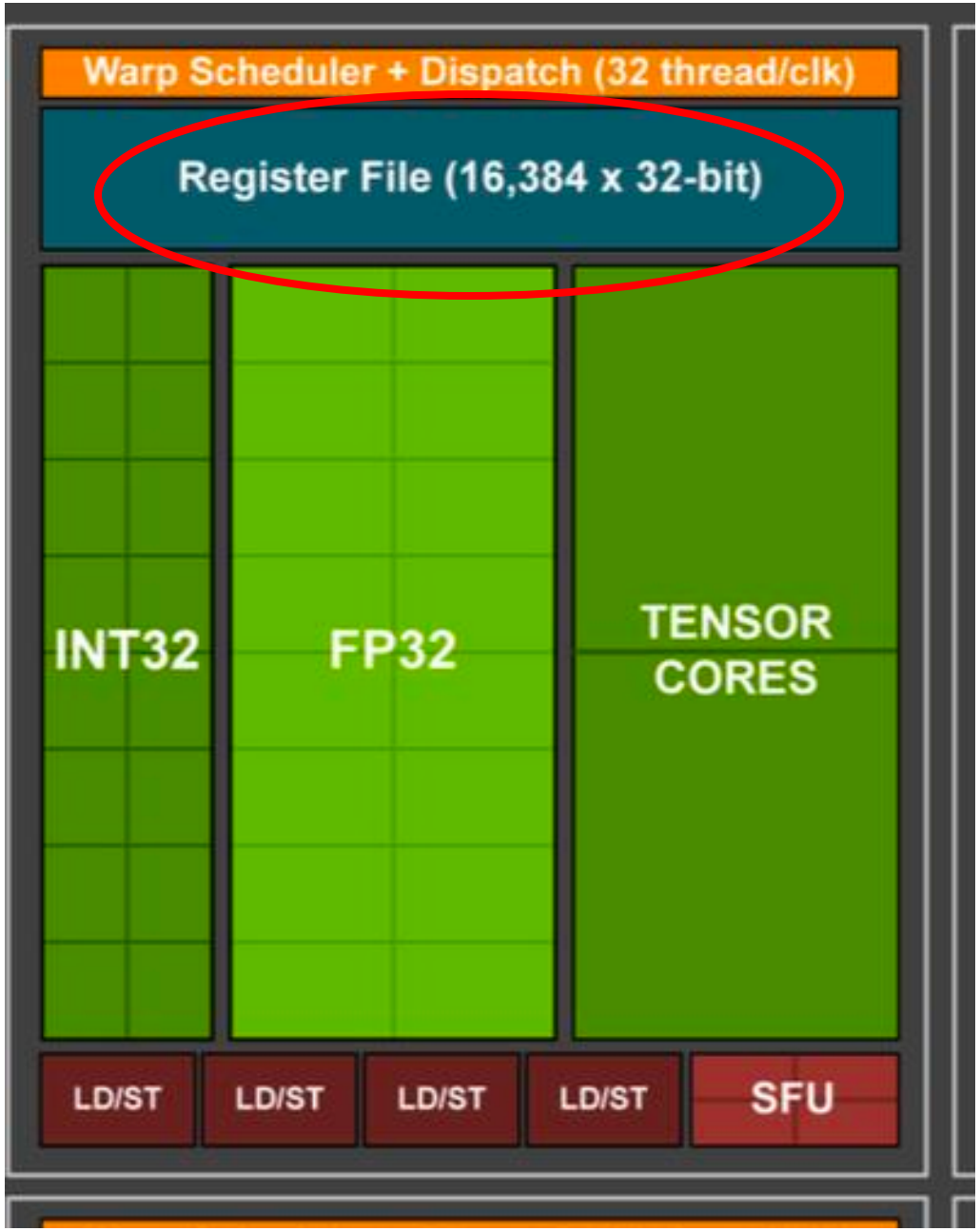
10 x 16 x 8 =
1280 ALUs.

Apple Silicon M2

https://www.apple.com/au/newsroom/2022/06/apple-unveils-m2-with-breakthrough-performance-and-capabilities/

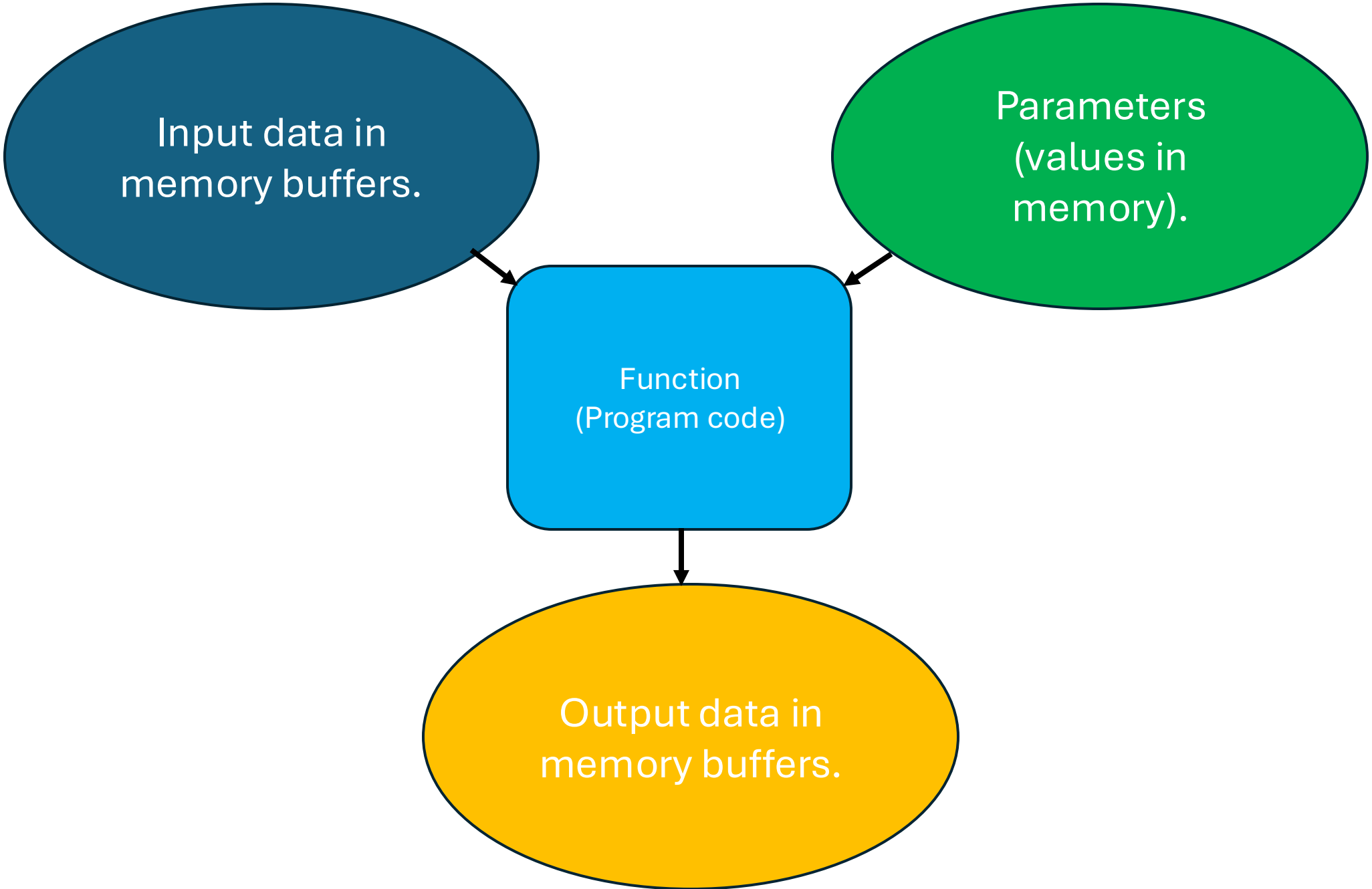
Nvidia TU102
Streaming multiprocessor (SM)
sub-partition (SMSP)

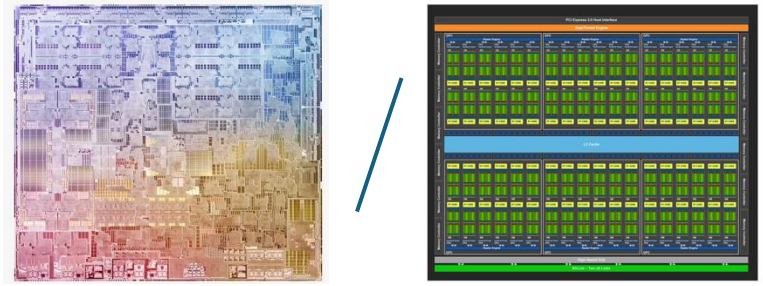
Runs threads in groups
of 32 at a time.
32 threads is a ‘Warp’.



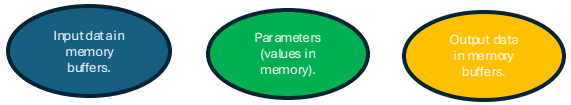
```
MTL::Device* Device = MTLCreateSystemDefaultDevice();
MTL::Library* Library = Device->newLibrary(NS::String::string("Compute.metallib",
                                                    UTF8StringEncoding),&ErrorPtr);
AdderFunction = Library->newFunction(NS::String::string("adder",UTF8StringEncoding));
MTL::Buffer* InputBuffer = Device->newBuffer(AllocationSize,BufferOptions);
MTL::Buffer* OutputBuffer = Device->newBuffer(AllocationSize,BufferOptions);
MTL::CommandQueue* CommandQueue = Device->newCommandQueue();
MTL::ComputePipelineState* PipelineState =
             Device->newComputePipelineState(AdderFunction,&ErrorPtr);
MTL::Size GridSize(Nx,Ny,1);
MTL::Size ThreadGroupDims(ThreadGroupSize / ThreadWidth,ThreadWidth,1);
MTL::CommandBuffer* CommandBuffer = CommandQueue->commandBuffer();
MTL::ComputeCommandEncoder* Encoder = CommandBuffer->computeCommandEncoder();
Encoder->setComputePipelineState(PipelineState);
Encoder->setBuffer(InputBuffer,0,1);
Encoder->setBuffer(OutputBuffer,0,2);
Encoder->dispatchThreads(GridSize,ThreadGroupDims);
Encoder->endEncoding();
CommandBuffer->commit();
CommandBuffer->waitUntilCompleted();
```

Metal ‘Adder’ code - skeleton


Input data in memory buffers.
Parameters (values in memory).
Function (Program code)
Output data in memory buffers.

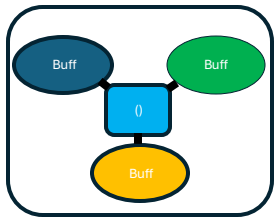


1 : Access GPU device



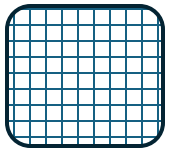
2 : Create memory buffers

3 : Create GPU program(s)



4 : Create Pipeline(s)

5: Set thread layout



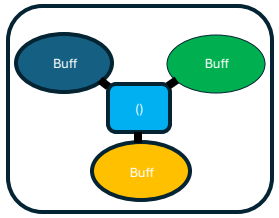
Q

6 : Create command queue(s)

CB

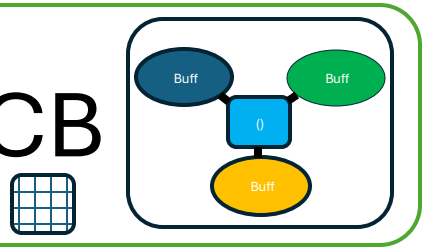
7 : Create Command buffer

CB



8 : Encode Command buffer

Q CB



9: Run command buffer

```
MTL::Device* Device = MTLCreateSystemDefaultDevice();
MTL::Library* Library = Device->newLibrary(NS::String::string("Compute.metallib",
                                                    UTF8StringEncoding),&ErrorPtr);
AdderFunction = Library->newFunction(NS::String::string("adder",UTF8StringEncoding));
MTL::Buffer* InputBuffer = Device->newBuffer(AllocationSize,BufferOptions);
MTL::Buffer* OutputBuffer = Device->newBuffer(AllocationSize,BufferOptions);
MTL::CommandQueue* CommandQueue = Device->newCommandQueue();
MTL::ComputePipelineState* PipelineState =
             Device->newComputePipelineState(AdderFunction,&ErrorPtr);
MTL::Size GridSize(Nx,Ny,1);
MTL::Size ThreadGroupDims(ThreadGroupSize / ThreadWidth,ThreadWidth,1);
MTL::CommandBuffer* CommandBuffer = CommandQueue->commandBuffer();
MTL::ComputeCommandEncoder* Encoder = CommandBuffer->computeCommandEncoder();
Encoder->setComputePipelineState(PipelineState);
Encoder->setBuffer(InputBuffer,0,1);
Encoder->setBuffer(OutputBuffer,0,2);
Encoder->dispatchThreads(GridSize,ThreadGroupDims);
Encoder->endEncoding();
CommandBuffer->commit();
CommandBuffer->waitUntilCompleted();
```

Metal ‘Adder’ code - skeleton

```
KVVulkanFramework Framework;
Framework.CreateVulkanInstance(StatusOK);
Framework.FindSuitableDevice(StatusOK);
Framework.CreateLogicalDevice(StatusOK);
InputBufferHndl = Framework.SetBufferDetails(C_InputBufferBinding,"STORAGE","SHARED",StatusOK);
Framework.CreateBuffer(InputBufferHndl,Length,StatusOK);
float* InputBufferAddr = (float*)Framework.MapBuffer(InputBufferHndl,&Bytes,StatusOK);
OutputBufferHndl = Framework.SetBufferDetails(C_OutputBufferBinding,"STORAGE",
                                                              "SHARED",StatusOK);
Framework.CreateBuffer(OutputBufferHndl,Length,StatusOK);
float* OutputBufferAddr = (float*)Framework.MapBuffer(OutputBufferHndl,&Bytes,StatusOK);
UniformBufferHndl = Framework.SetBufferDetails(C_UniformBufferBinding,
                                                 "UNIFORM","SHARED",StatusOK);
Framework.CreateBuffer(UniformBufferHndl,SizeInBytes,StatusOK);
void* UniformBufferAddr = Framework.MapBuffer(UniformBufferHndl,&Bytes,StatusOK);
Framework.CreateVulkanDescriptorSetLayout(Handles,&SetLayout,StatusOK);
Framework.CreateVulkanDescriptorPool(Handles,1,&DescriptorPool,StatusOK);
Framework.AllocateVulkanDescriptorSet(SetLayout,DescriptorPool,&DescriptorSet,StatusOK);
Framework.SetupVulkanDescriptorSet(Handles,DescriptorSet,StatusOK);
Framework.CreateComputePipeline("Adder.spv","main",
                        &SetLayout,&ComputePipelineLayout,&ComputePipeline,StatusOK);
Framework.GetDeviceQueue(&ComputeQueue,StatusOK);
Framework.CreateCommandPool(&CommandPool,StatusOK);
Framework.CreateComputeCommandBuffer(CommandPool,&CommandBuffer,StatusOK);
Framework.RecordComputeCommandBuffer(CommandBuffer,ComputePipeline,
                                  ComputePipelineLayout,&DescriptorSet,WorkGroupCounts,StatusOK);
Framework.RunCommandBuffer(ComputeQueue,CommandBuffer,StatusOK);
```

Vulkan ‘Adder’ code - skeleton

Image 1024 x 512 pixels

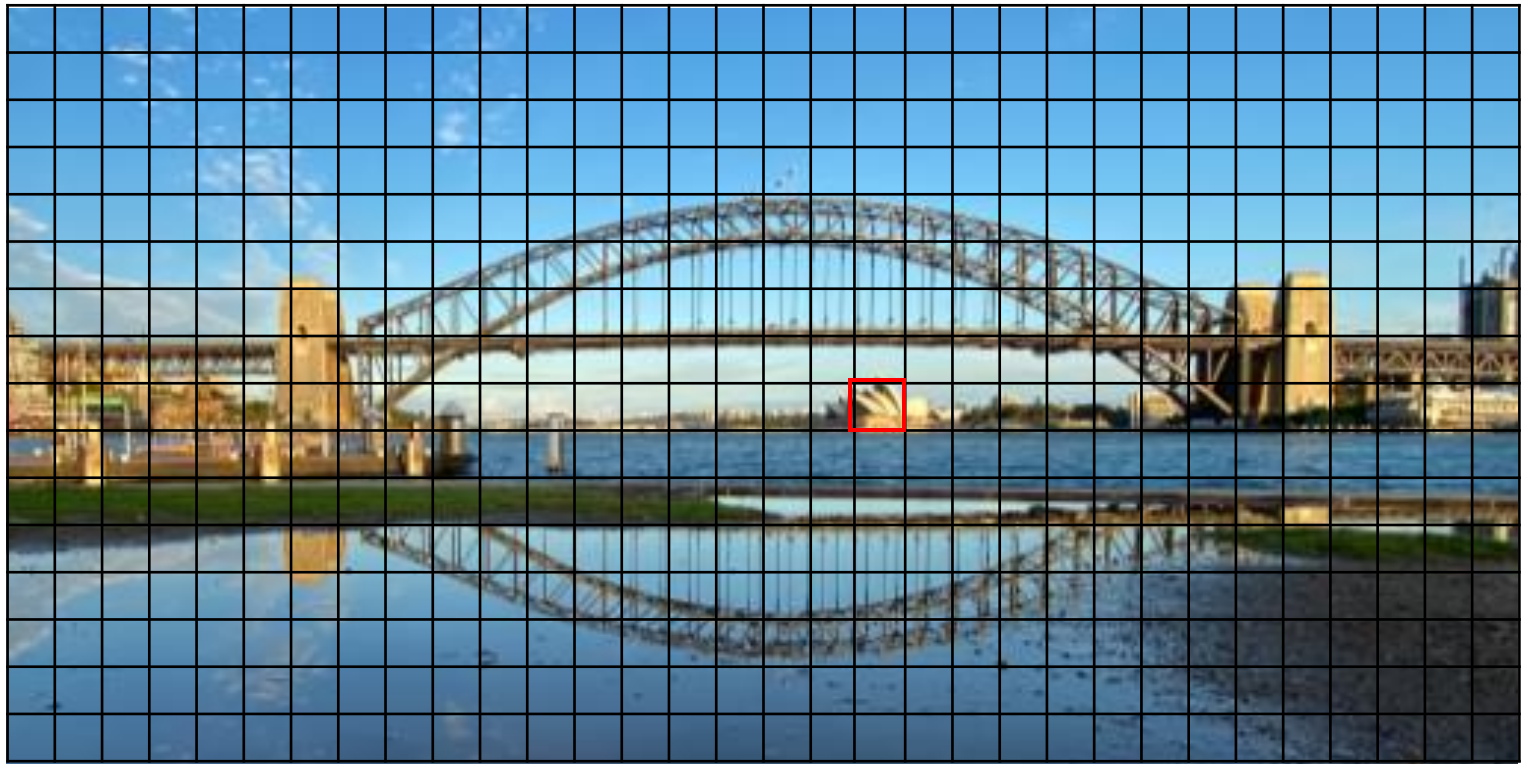
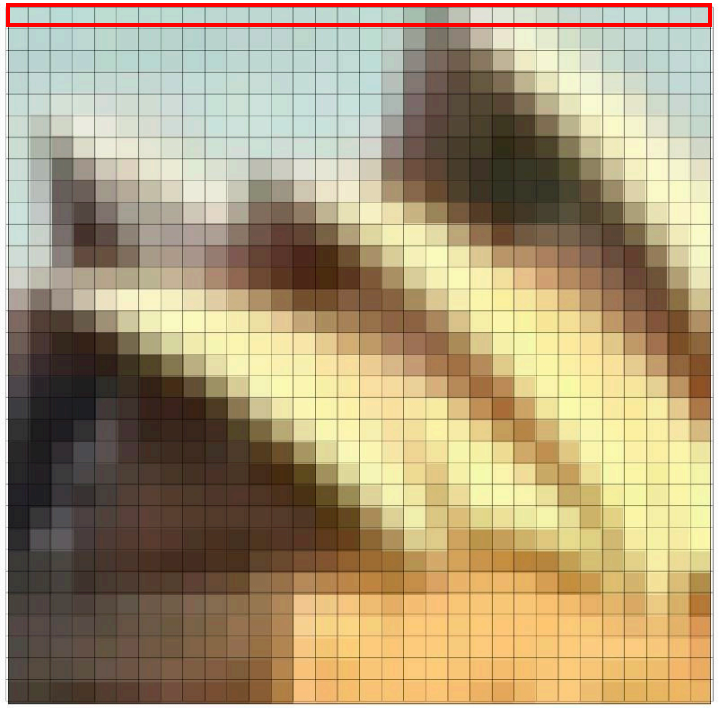


Image 1024 x 512 pixels

Split into 32 x 16 ‘threadgroups’,
each of 32 x 32 threads, in this case.

Each thread in a threadgroup
works on one image pixel, in
this case.

Threads in a threadgroup
execute in ‘warps’ of 32 threads
running together in lockstep.



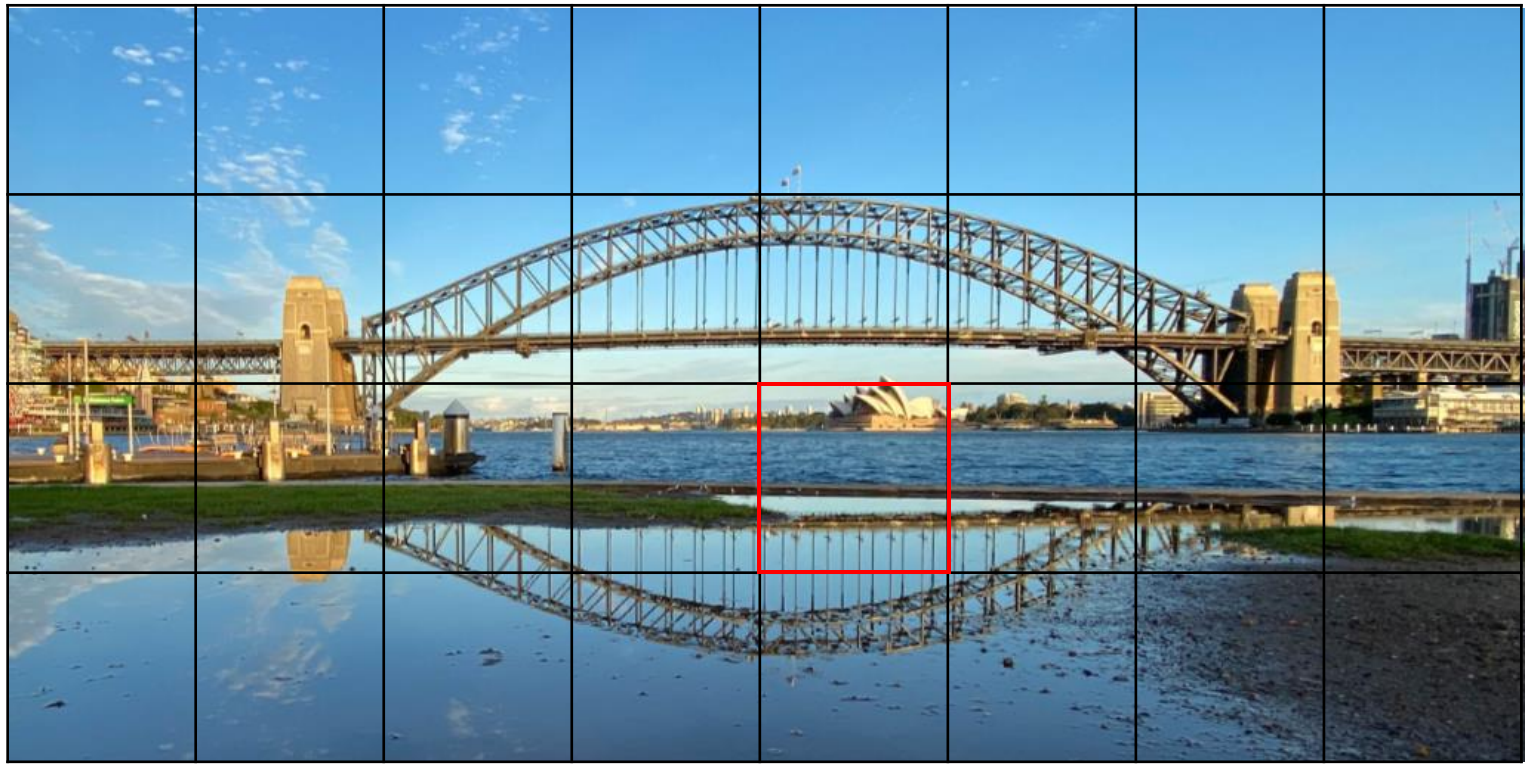
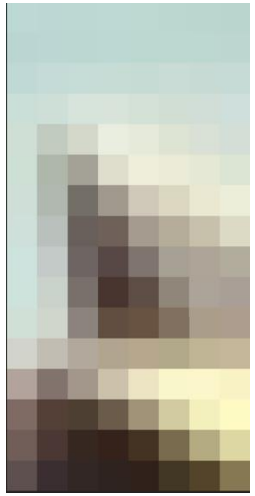


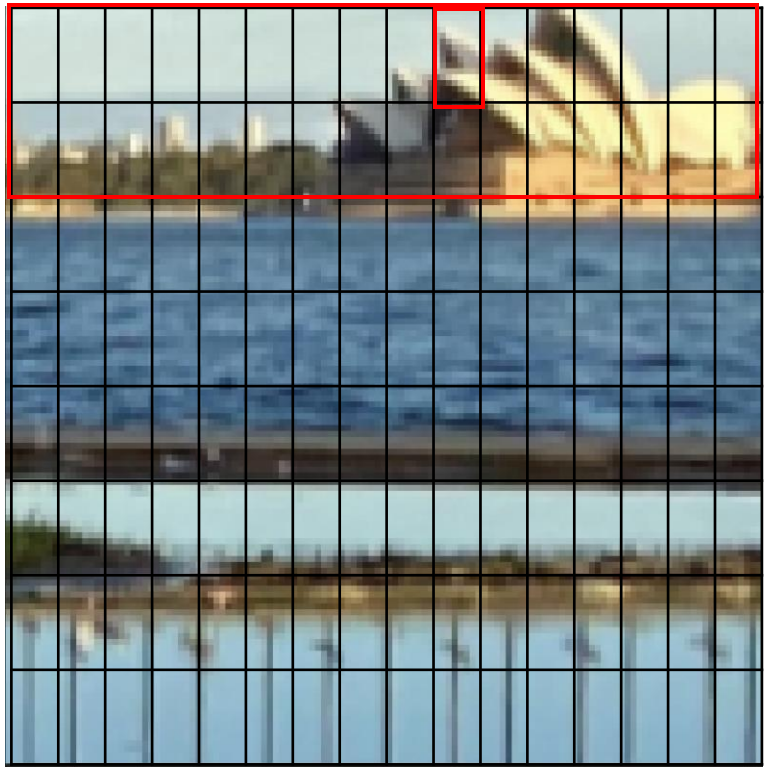
Image 1024 x 512 pixels

Split into 8 x 4 ‘threadgroups’,
each of 16 x 8 threads, in this case.

In this case, **each** thread works on a grid of 8 x 16 pixels.



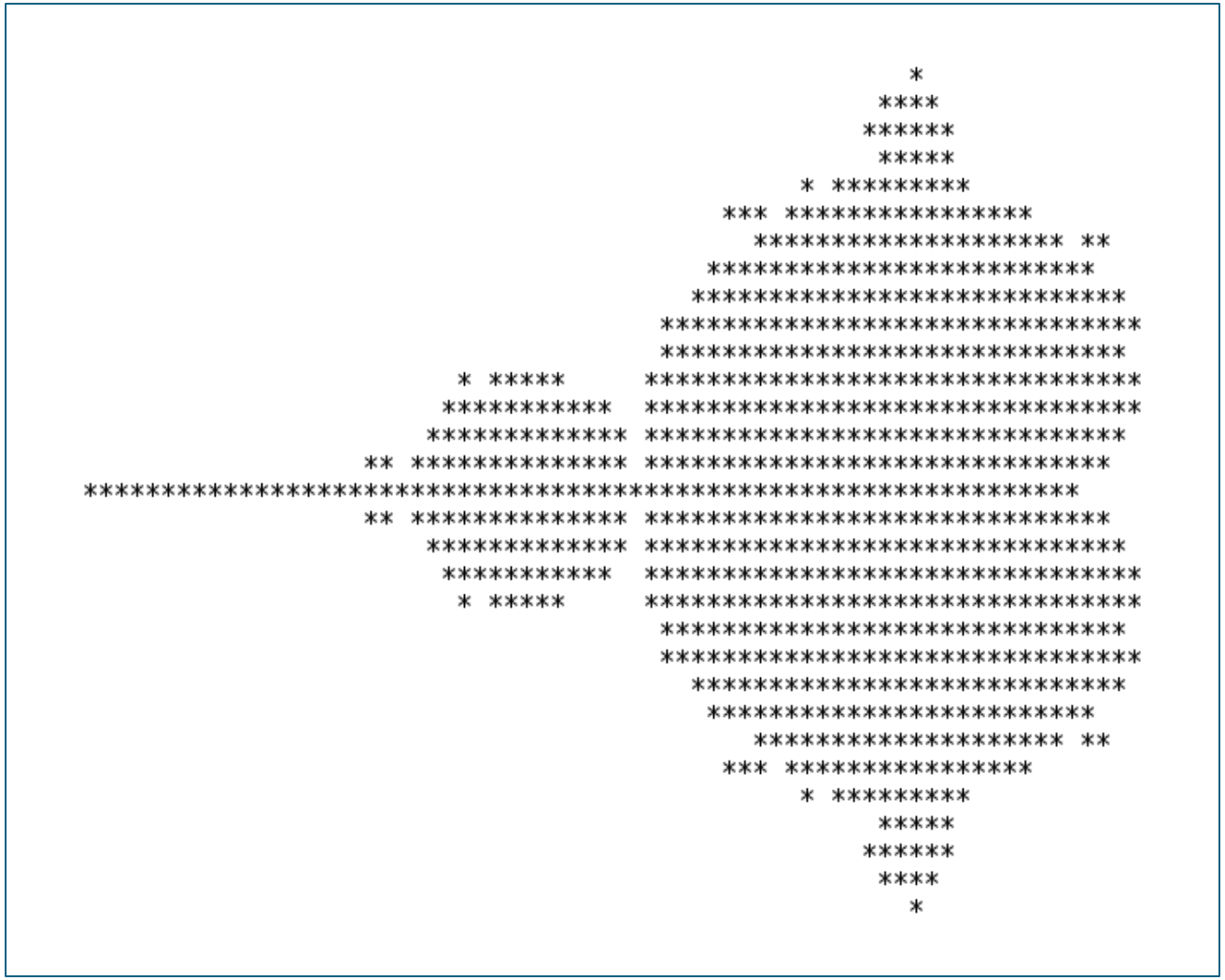
OR...



Threads in a threadgroup execute in ‘warps’ of 32 threads running together in lockstep.

```
float CalcMedian(float X[NPIXSQ_MAX], uint len)
{
   #define swap(a,b) {float t = X[a]; X[a] = X[b], X[b] = t;}
   uint left = 0, right = len - 1;
   uint cent = len/2;
   float pivot;
   uint pos, i;
   float median = 0.0;

   while (left < right) {
      pivot = X[cent];
      swap(cent, right);
      for (i = pos = left; i < right; i++) {
         if (X[i] < pivot) {
            swap(i, pos);
            pos++;
         }
      }
      swap(right, pos);
      if (pos == cent) break;
      if (pos < cent) left = pos + 1;
      else right = pos - 1;
   }
   median = X[cent];
```



The first published picture of the Mandelbrot set, by Robert W. Brooks and Peter Matelski in 1978

By Elphaba - Own work, Public Domain,
https://commons.wikimedia.org/w/index.php?curid=1627040