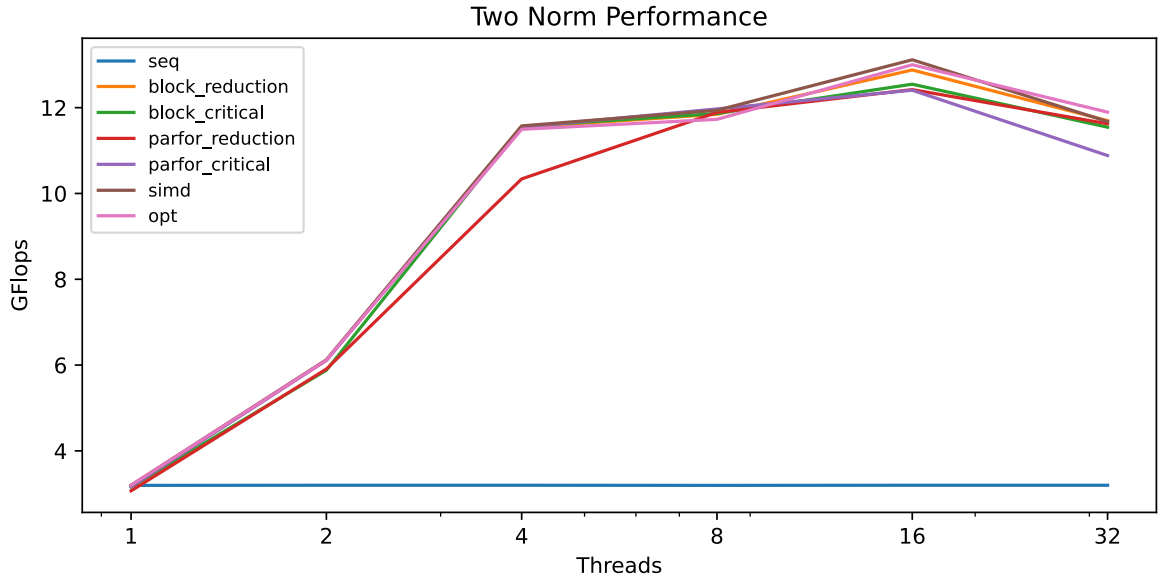
CSE P 524
Parallel Computation

Lecture 9
GPUs, CUDA, Thrust

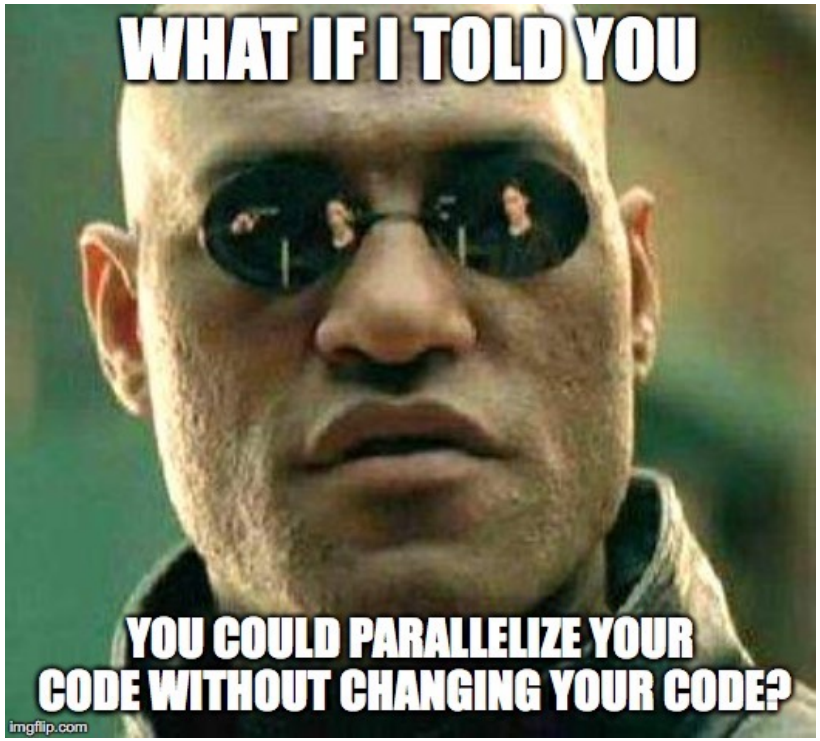
Andrew Lumsdaine
Northwest Institute for Advanced Computing
Pacific Northwest National Laboratory
University of Washington
Seattle, WA

OpenMP Performance

- Run nbody.exe on mcu
- The compiler matters



What if I told you

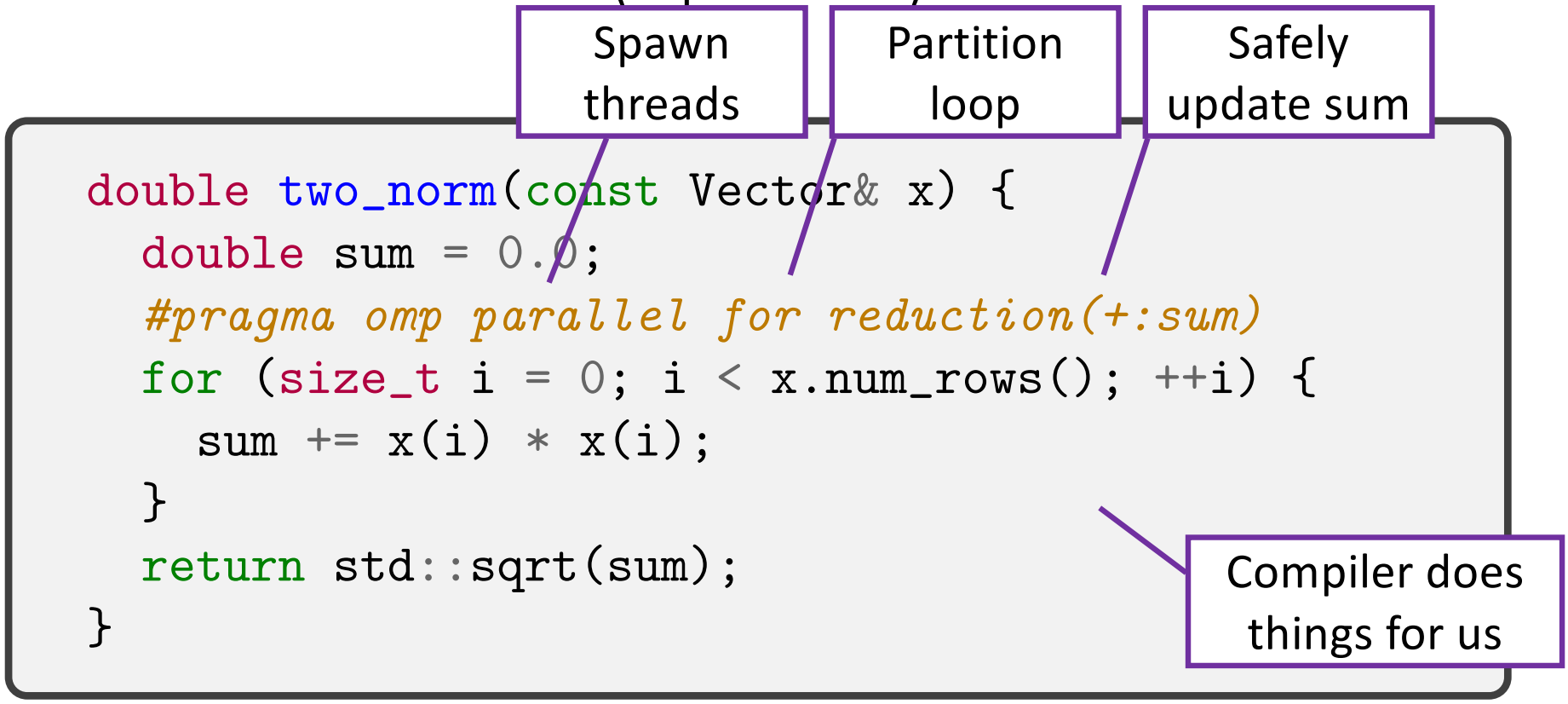



```
double two_norm(const Vector& x) {
  double sum = 0.0;
  for (size_t i = 0; i < x.num_rows(); ++i) {
    sum += x(i) * x(i);
  }
  return std::sqrt(sum);
}
```

This does not change

OpenMP

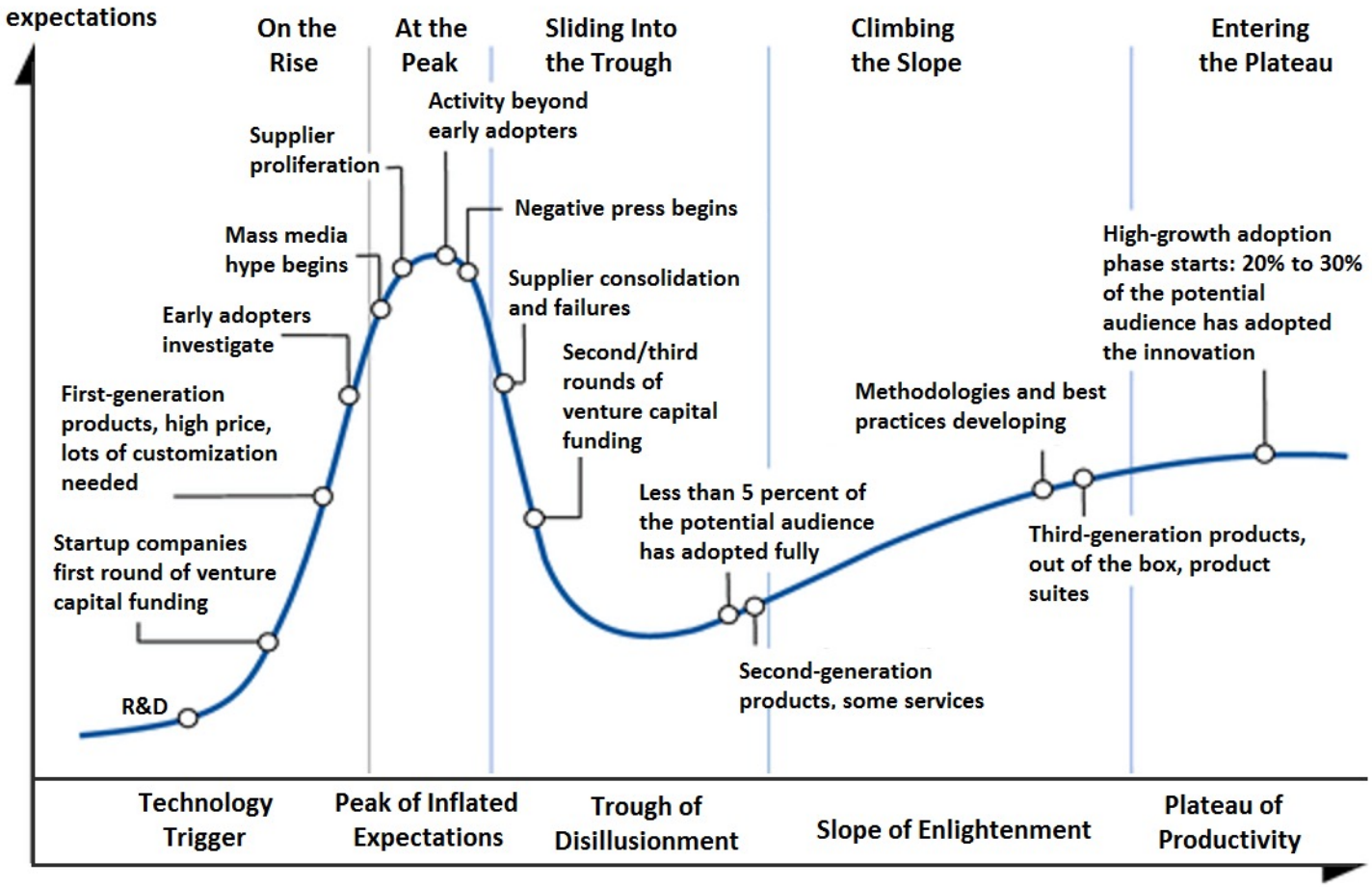
Two Norm Function (Open MP)



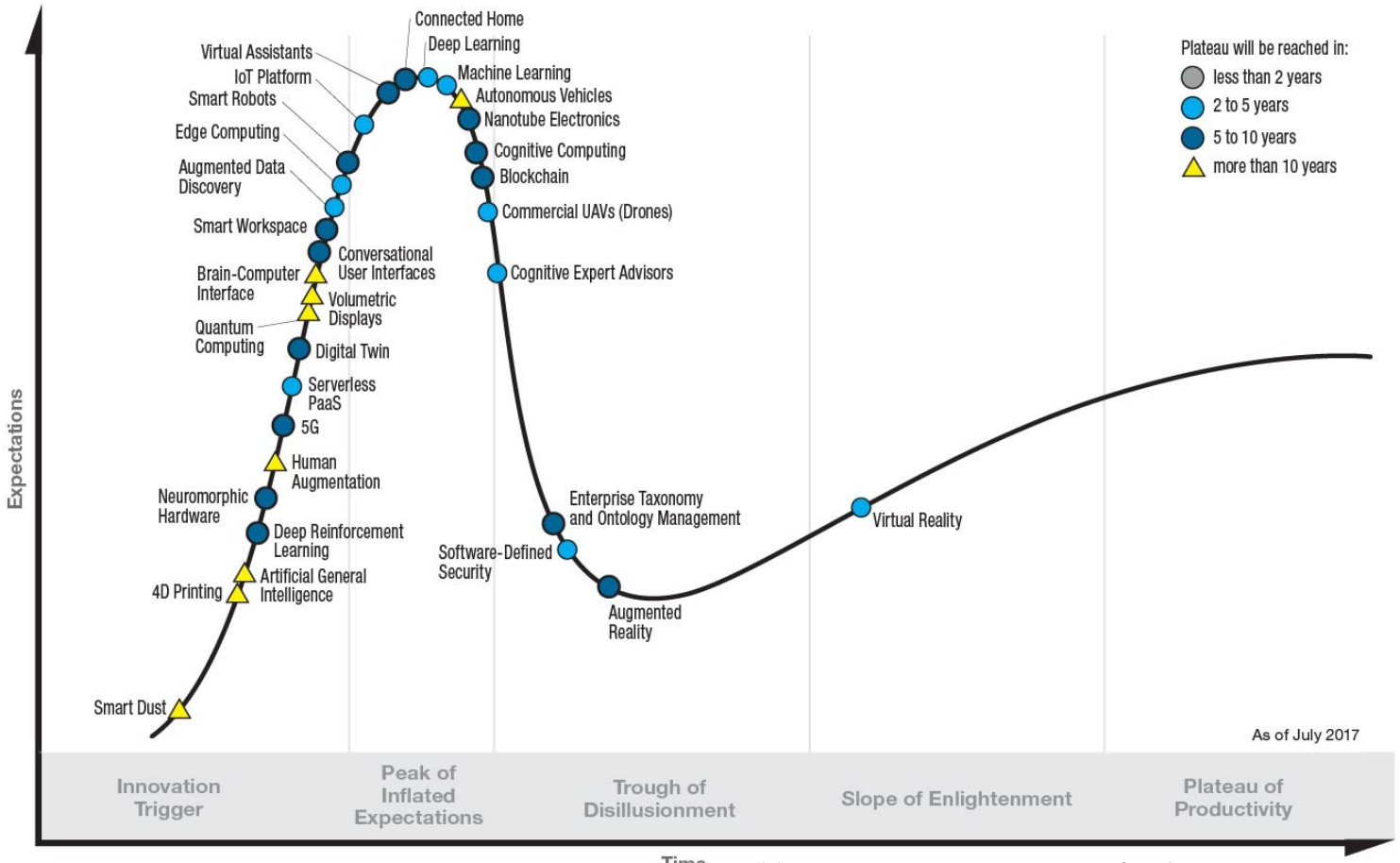
#pragma omp simd

```
int main(int argc, char* argv[]) {
  size_t intervals          = 1024 * 1024;
  if (argc >= 2) intervals = std::stol(argv[1]);
  double h                  = 1.0 / (double)intervals;

  double pi = 0.0;

  #pragma omp parallel for simd reduction(+:pi)
  for (size_t i = 0; i < intervals; ++i) {
    pi += (h * 4.0) / (1.0 + (i * h * i * h));
  }

  std::cout << "pi is approximately " << std::setprecision(15) << pi << std::endl;
  std::cout << "error is " << std::abs(PI25DT - pi) << std::endl;

  return 0;
}
```

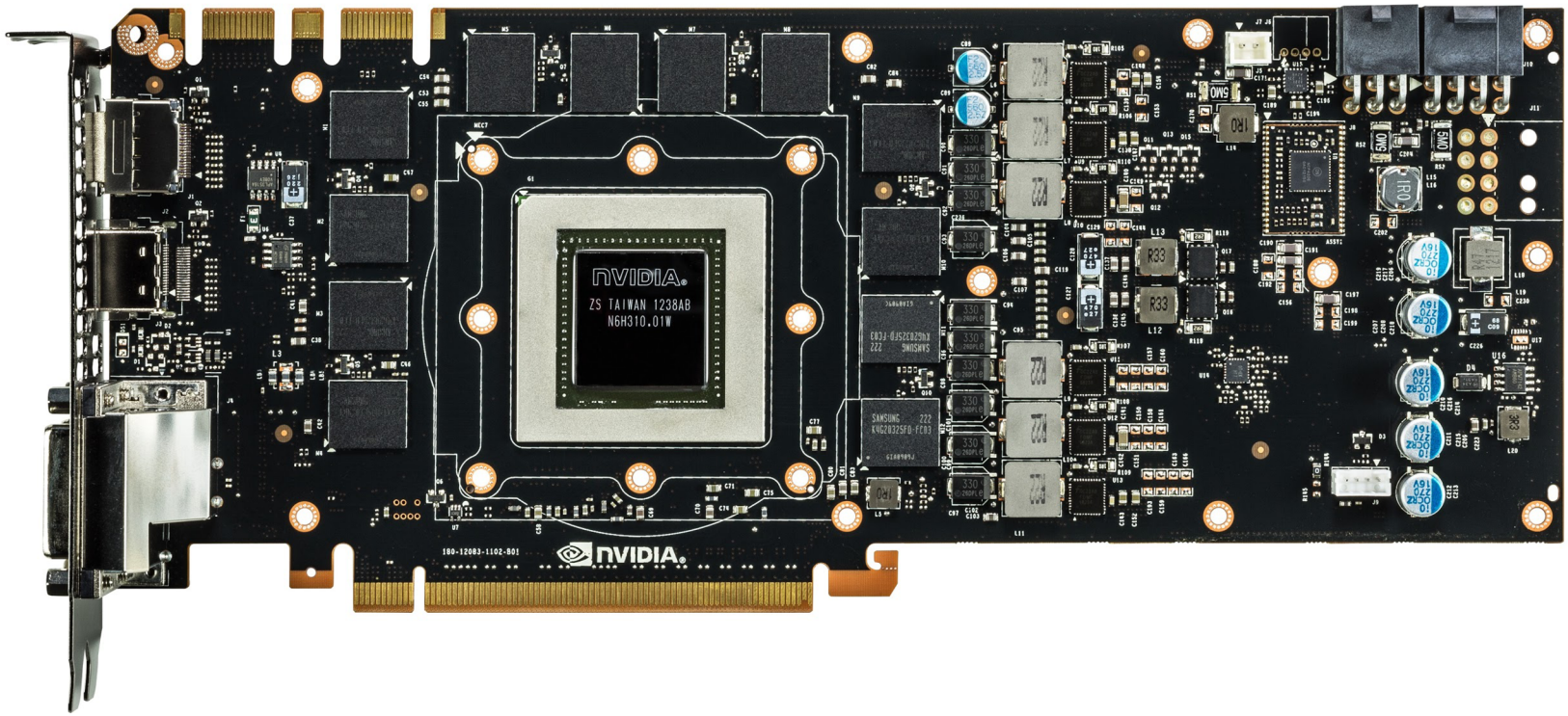
The Hype Cycle



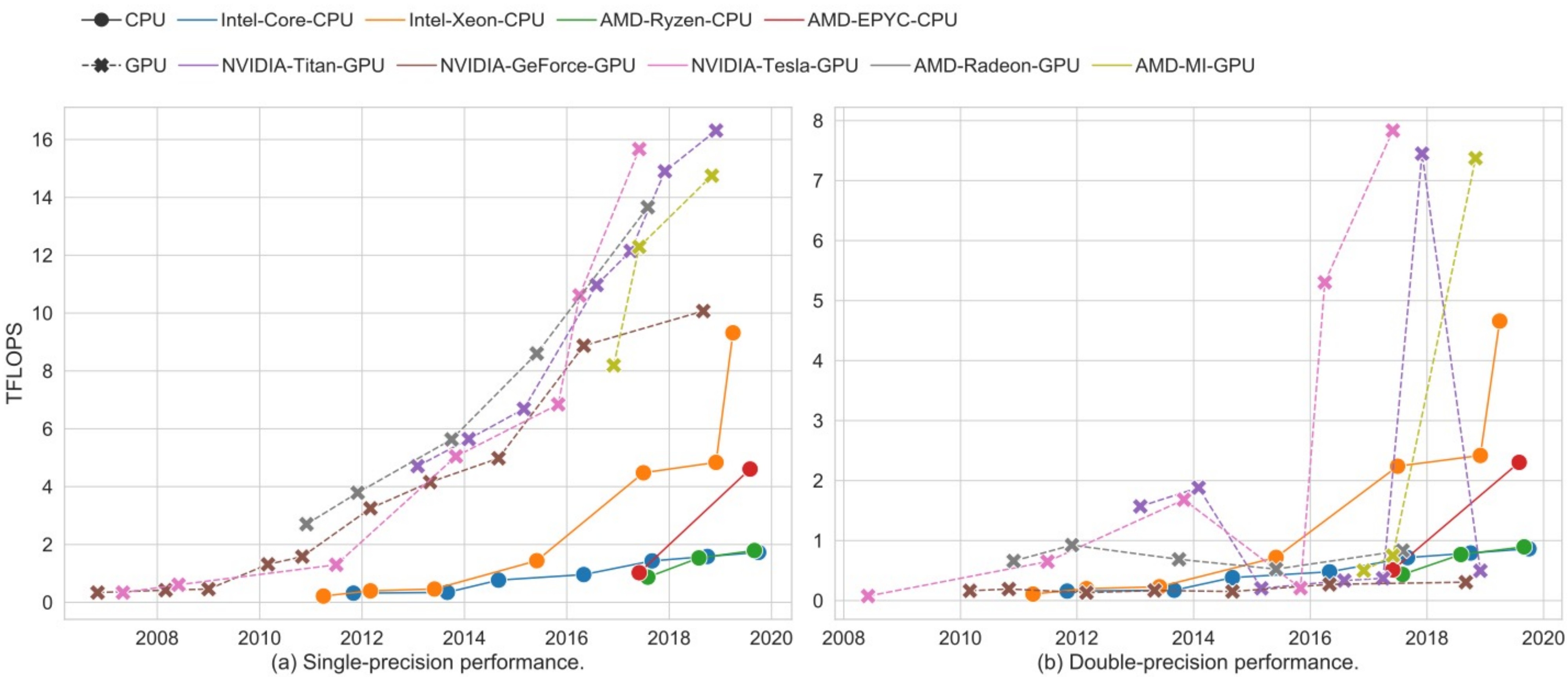
Gartner Hype Cycle



GPU

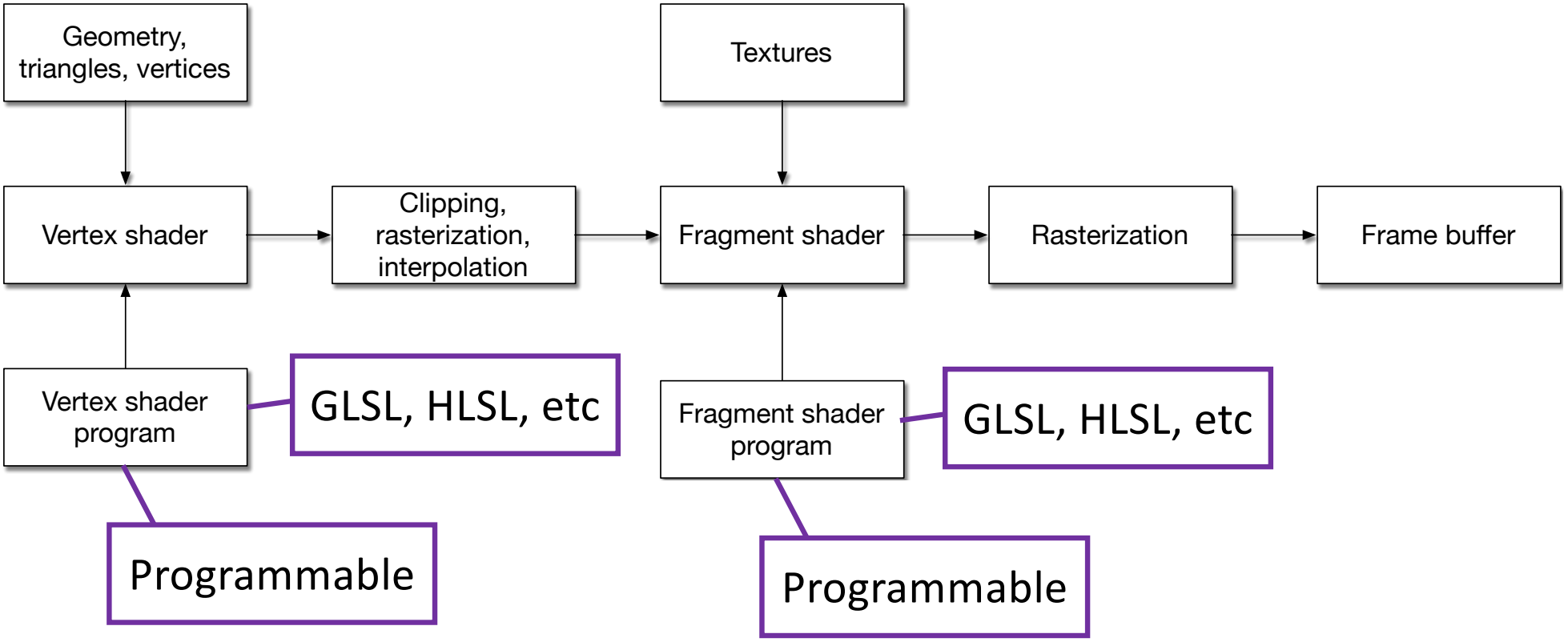

NVIDIA
ZS TAIWAN 1238AB
N6H310.01W
180-12083-1102-B01
NVIDIA

Performance

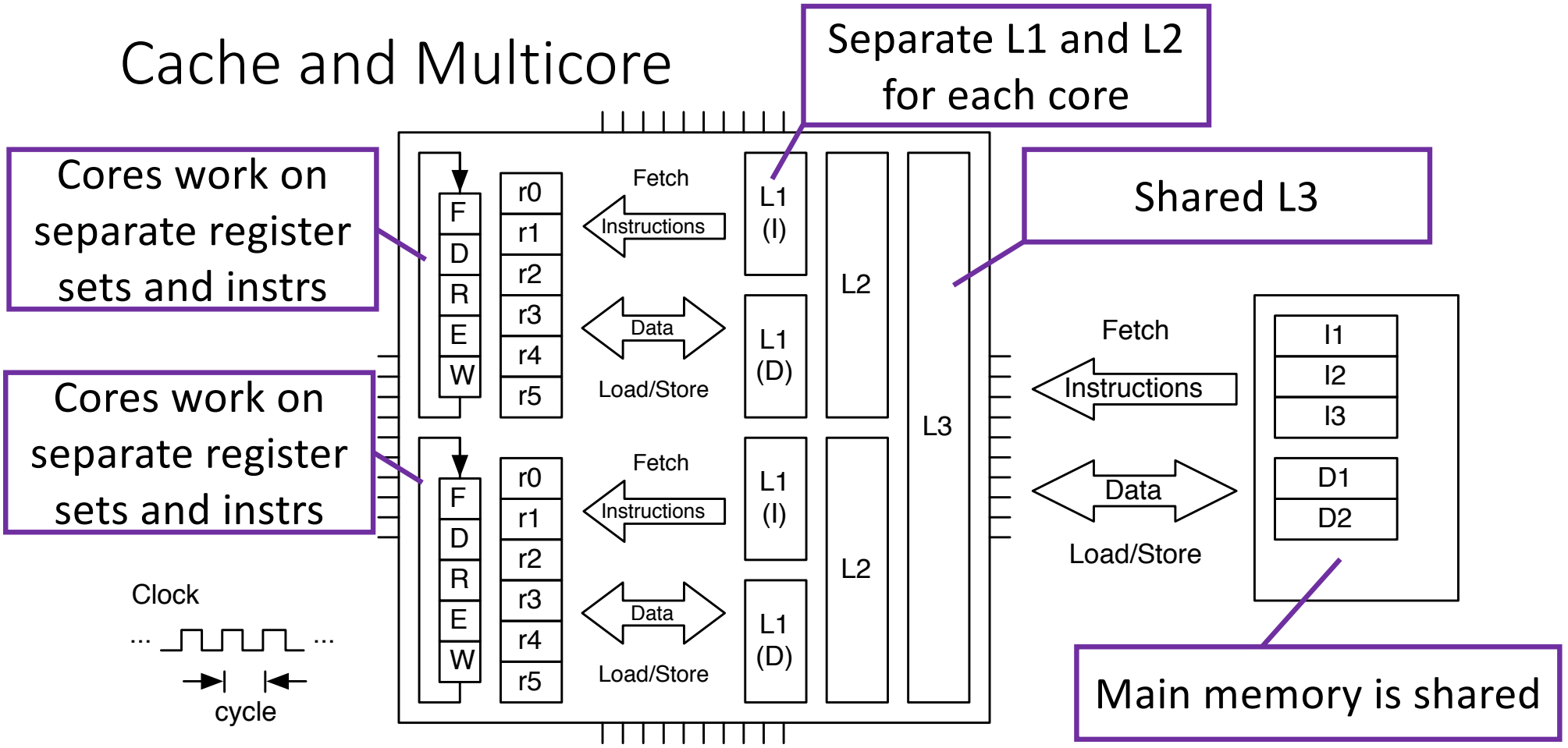


Summarizing CPU and GPU design trends with product data. Yifan Sun, et al. 2019

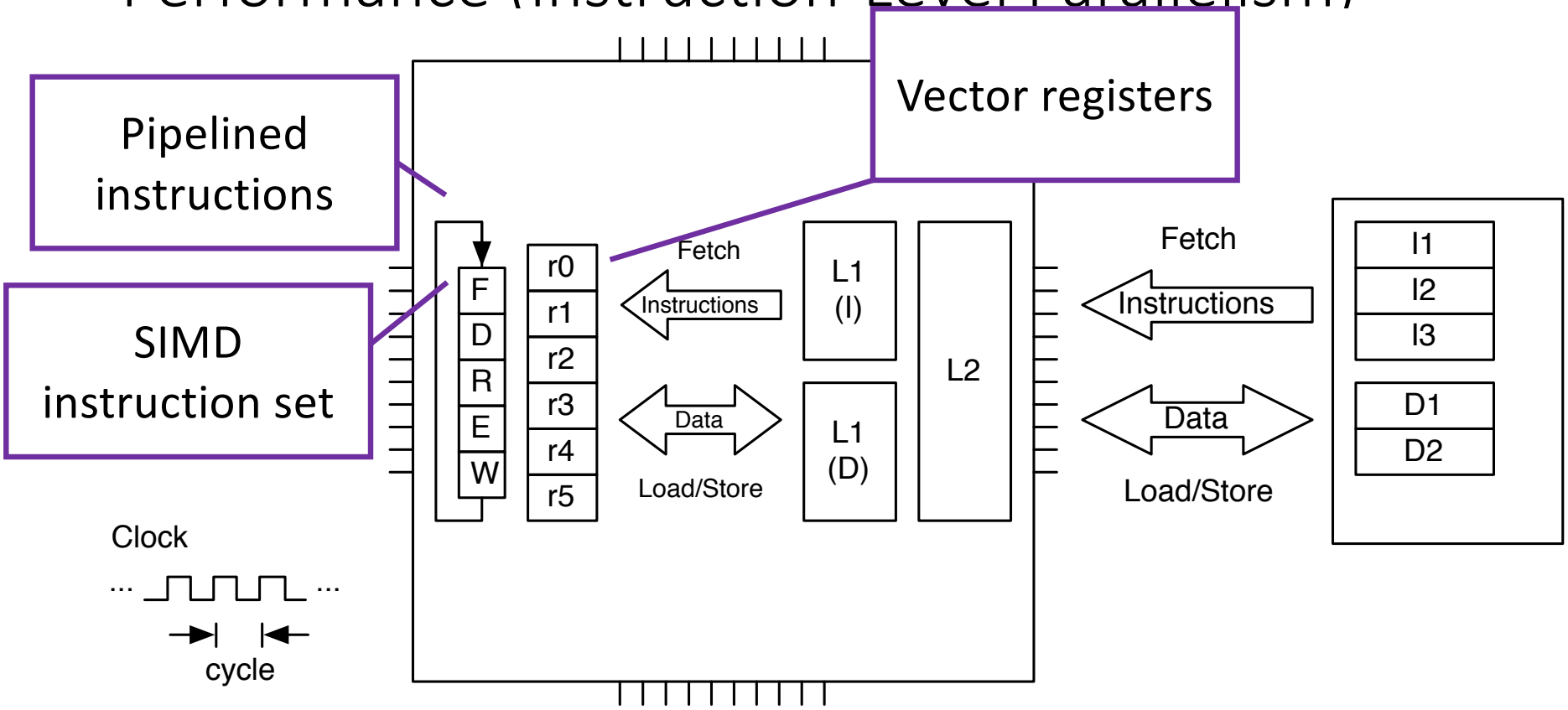
Graphics Pipeline



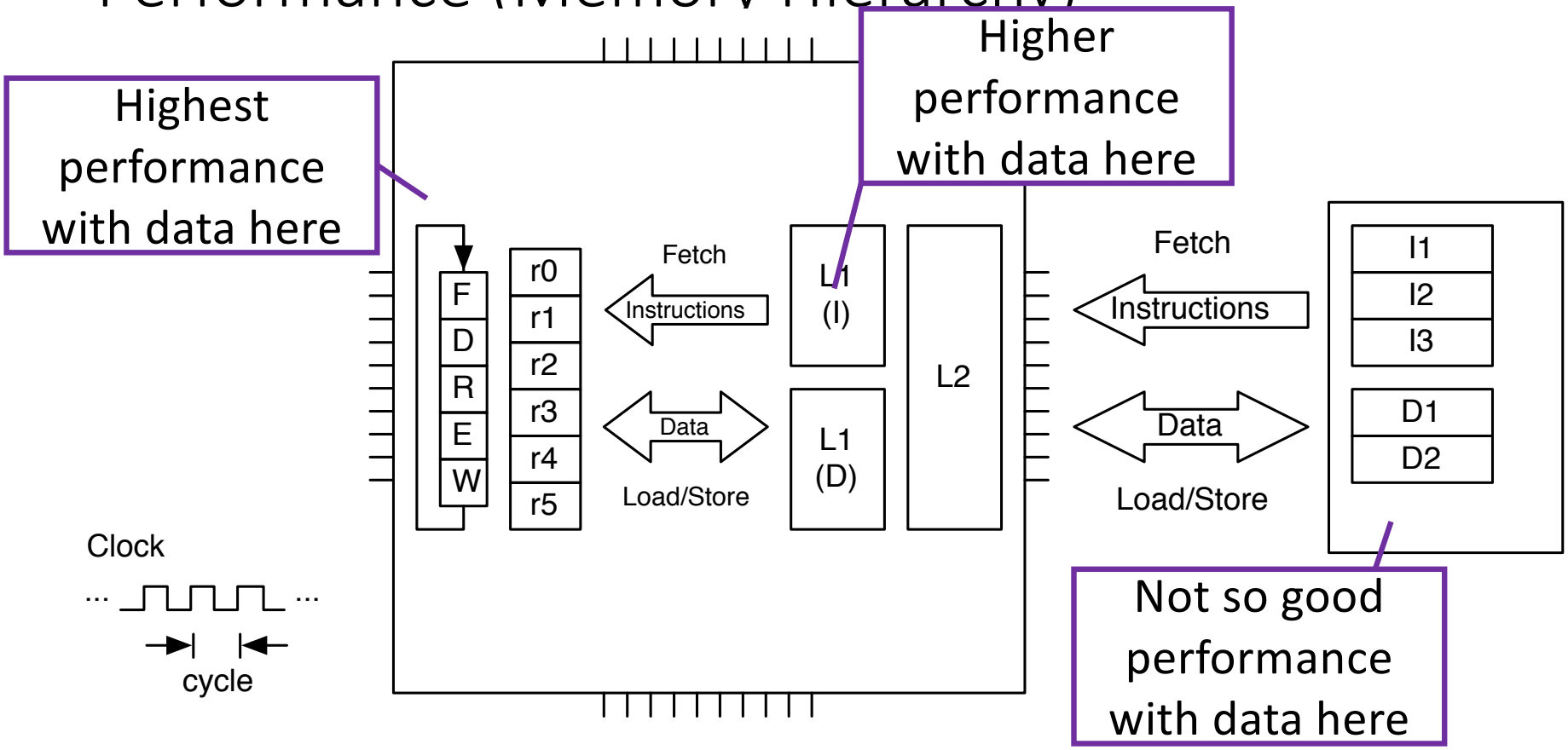
Cache and Multicore



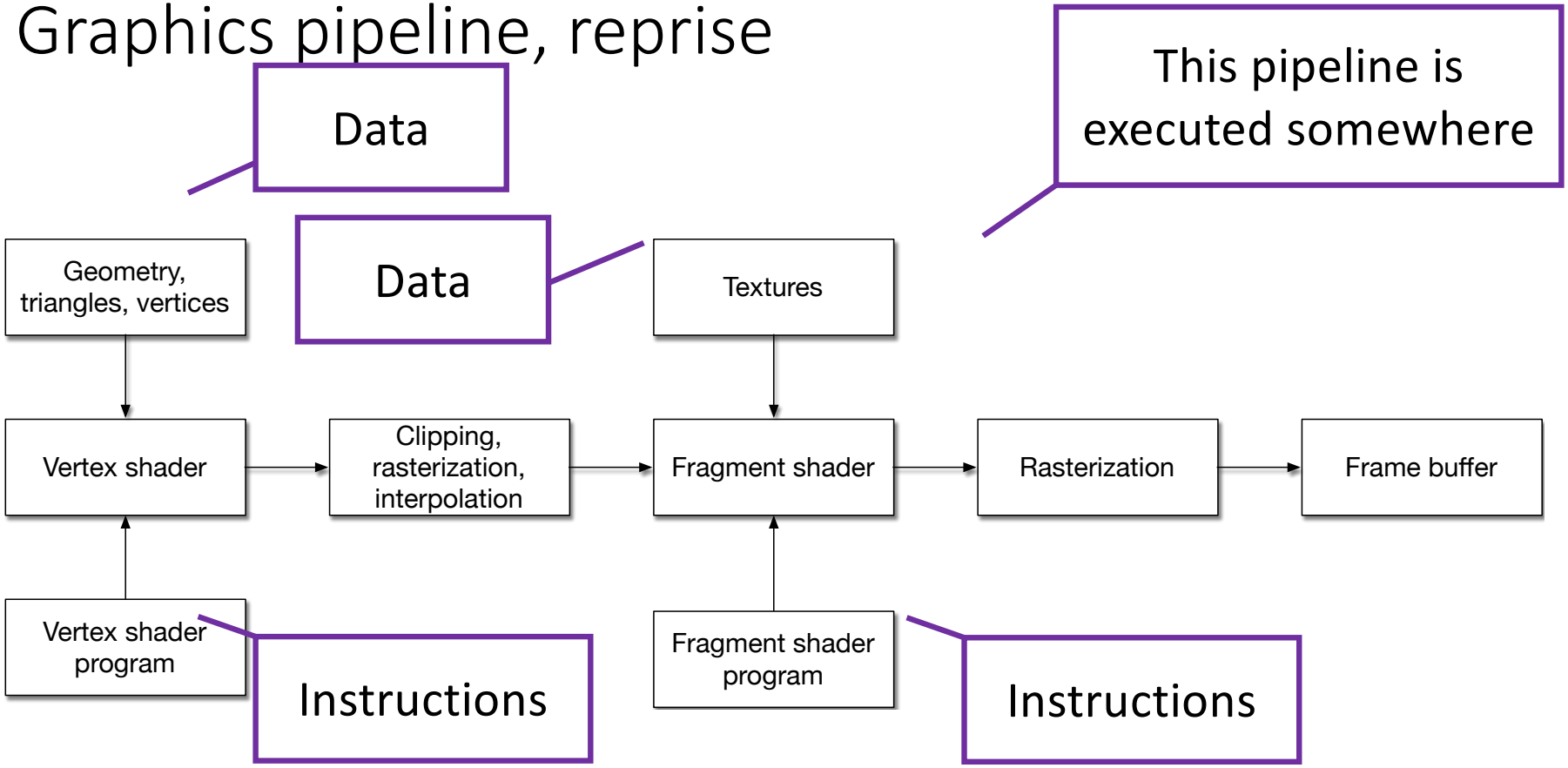
Performance (Instruction-Level Parallelism)



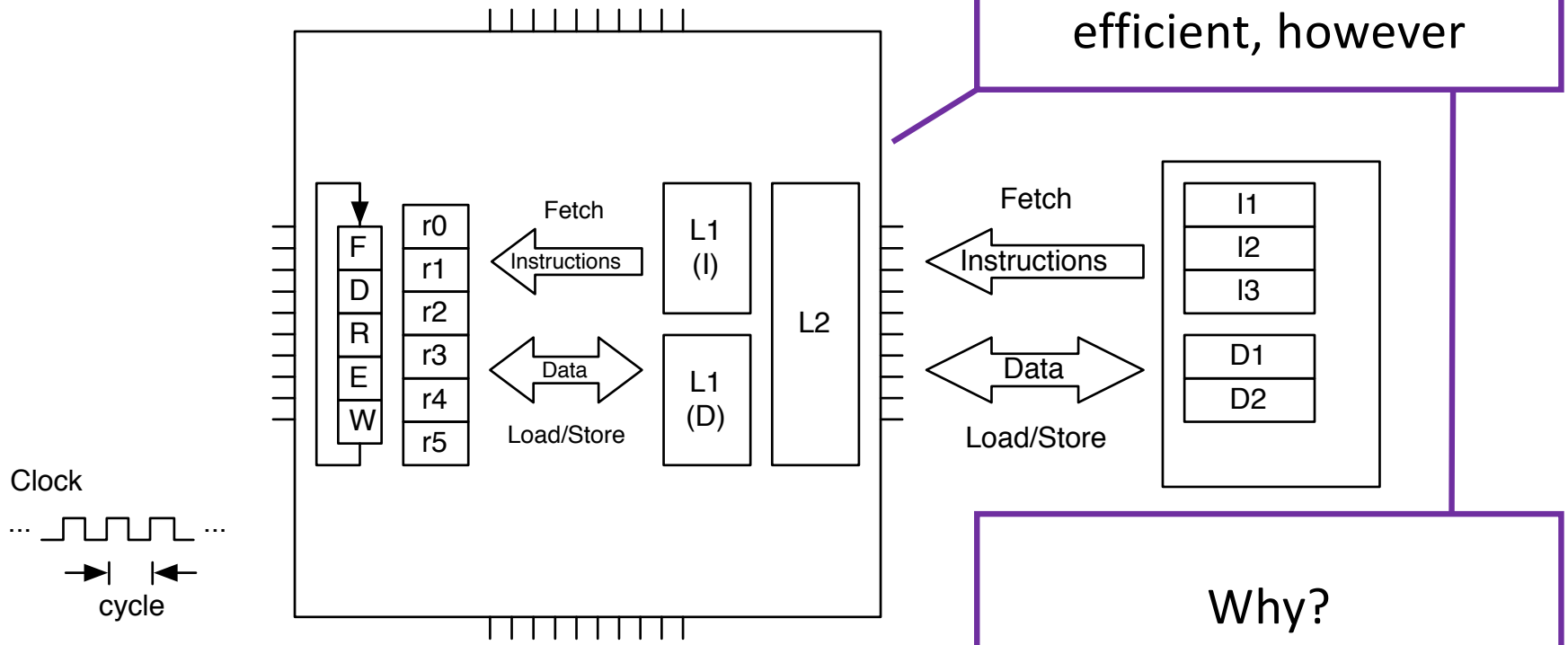
Performance (Memory Hierarchy)



Graphics pipeline, reprise



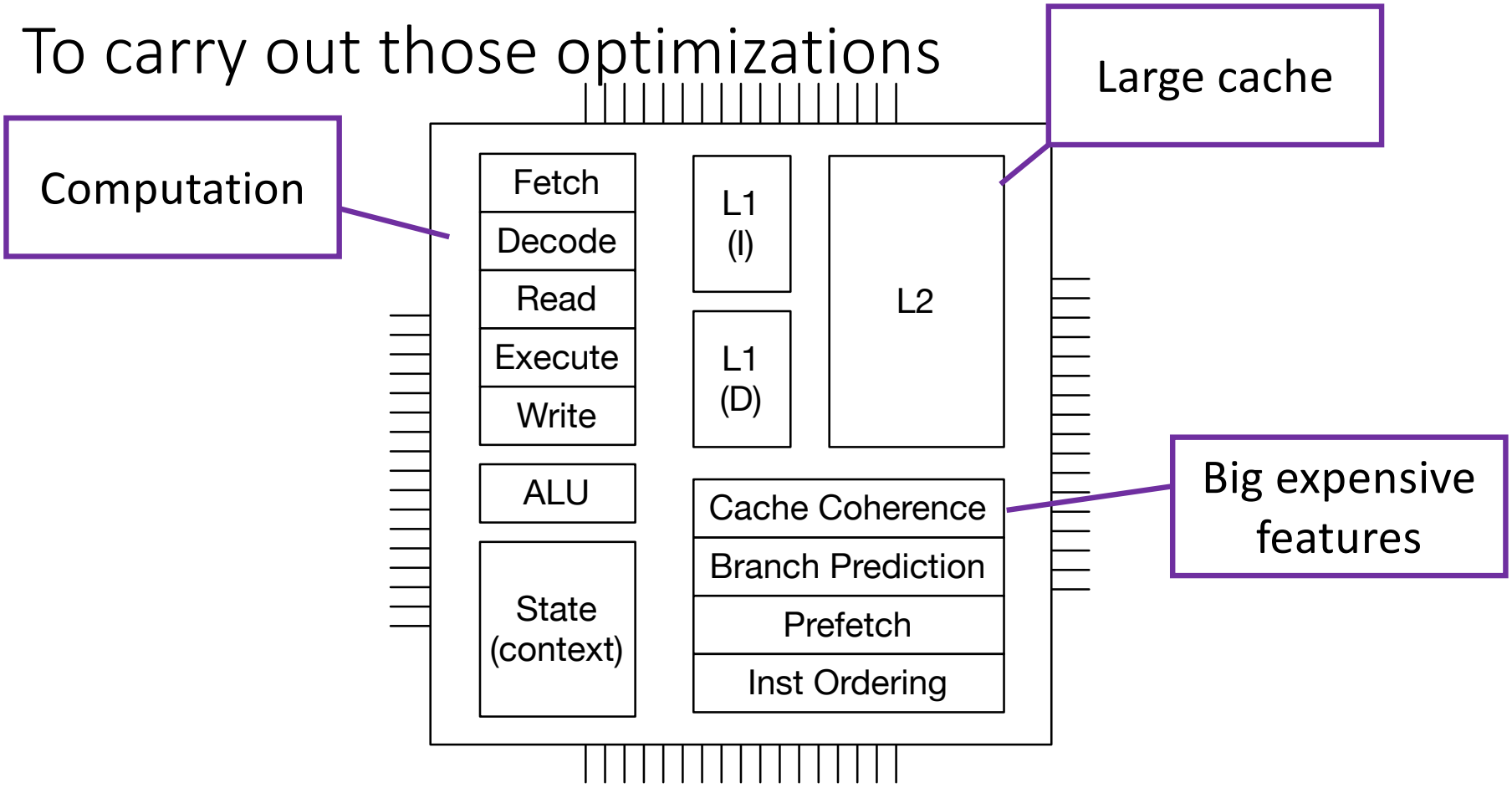
Pipeline can be executed on CPU



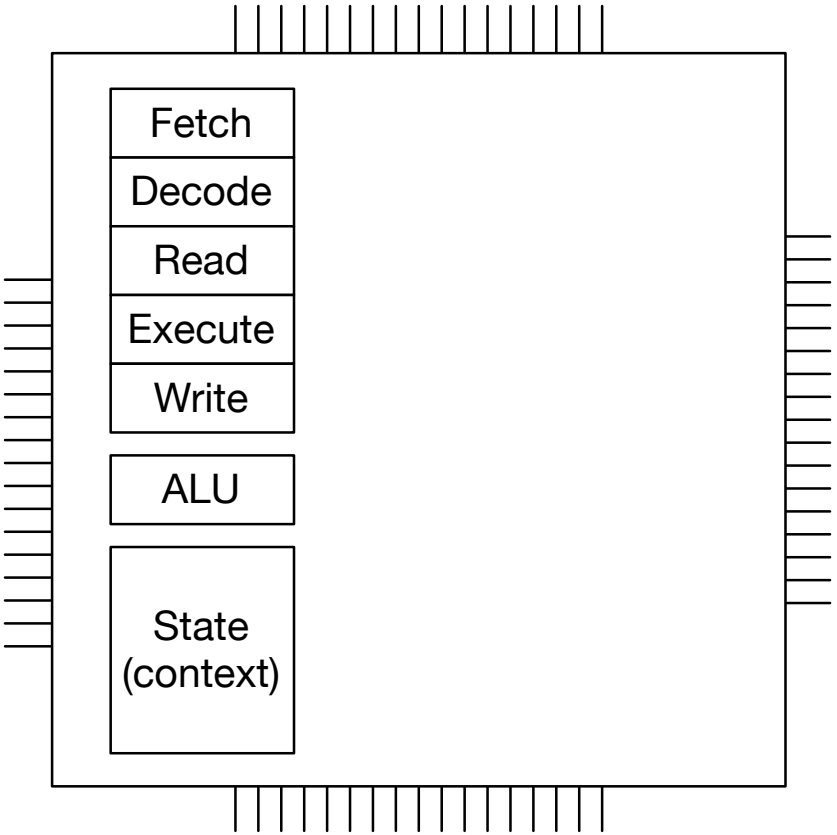
For what are CPUs optimized?

- CPUs are optimized for _____

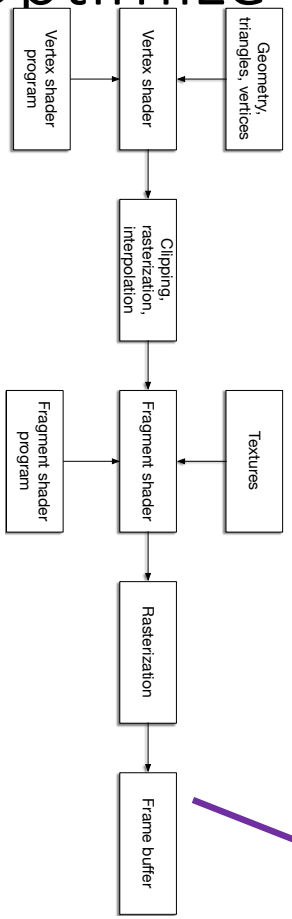
To carry out those optimizations

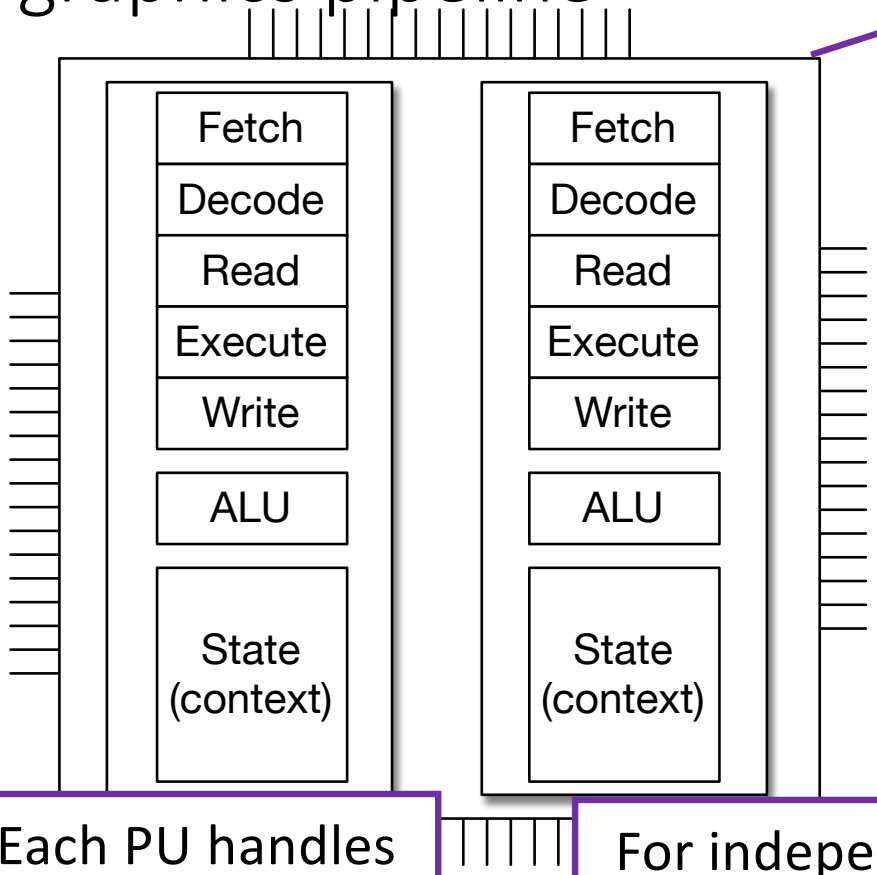


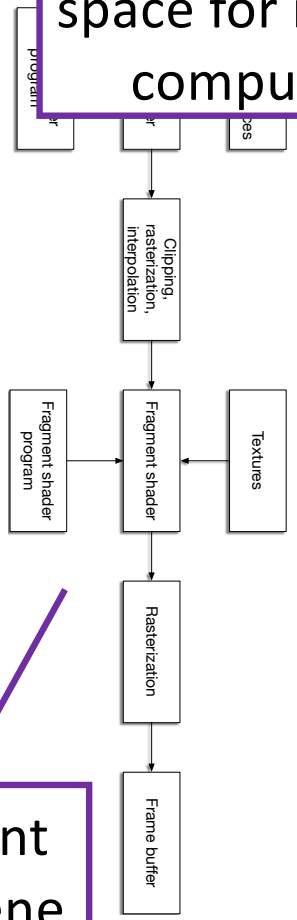
Optimize instead for graphics pipeline



Optimize for graphics pipeline





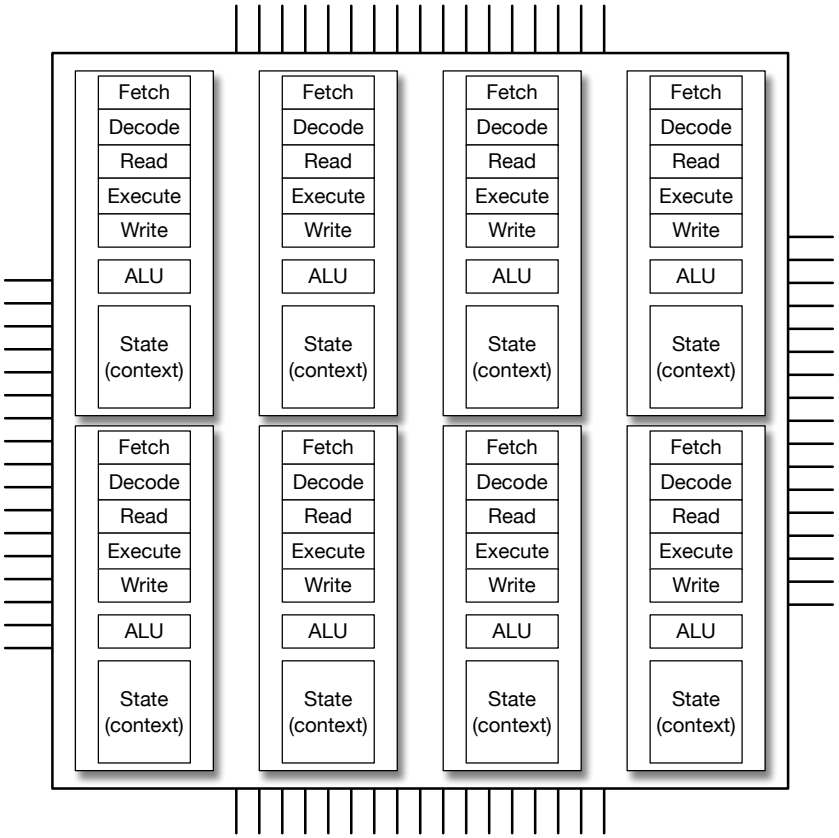


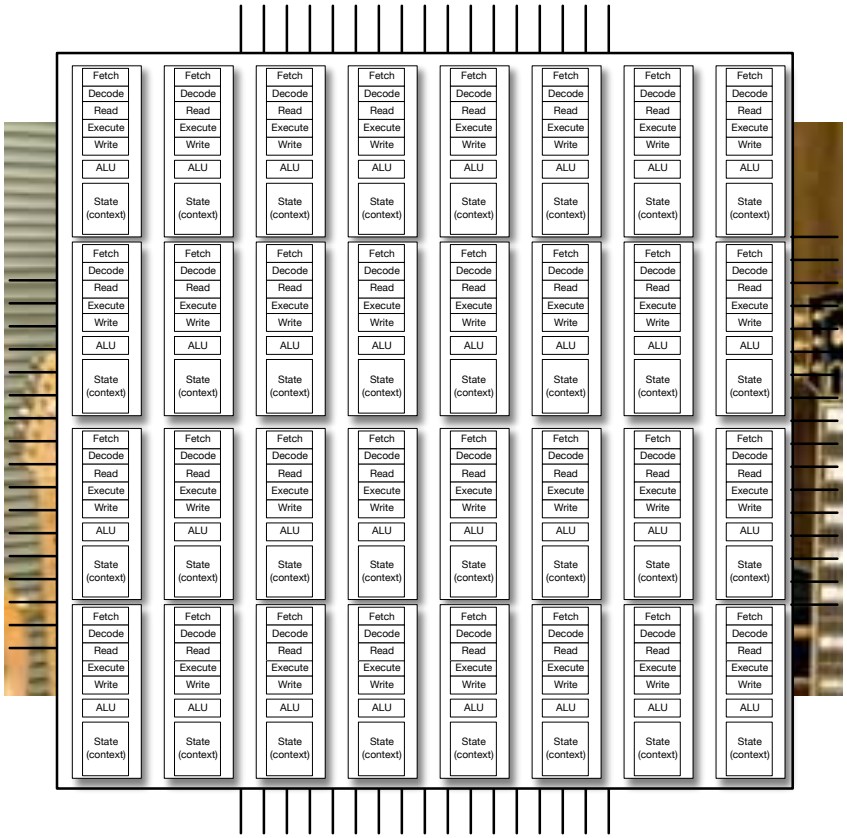
Use freed up space for more compute

Each PU handles different pipelines

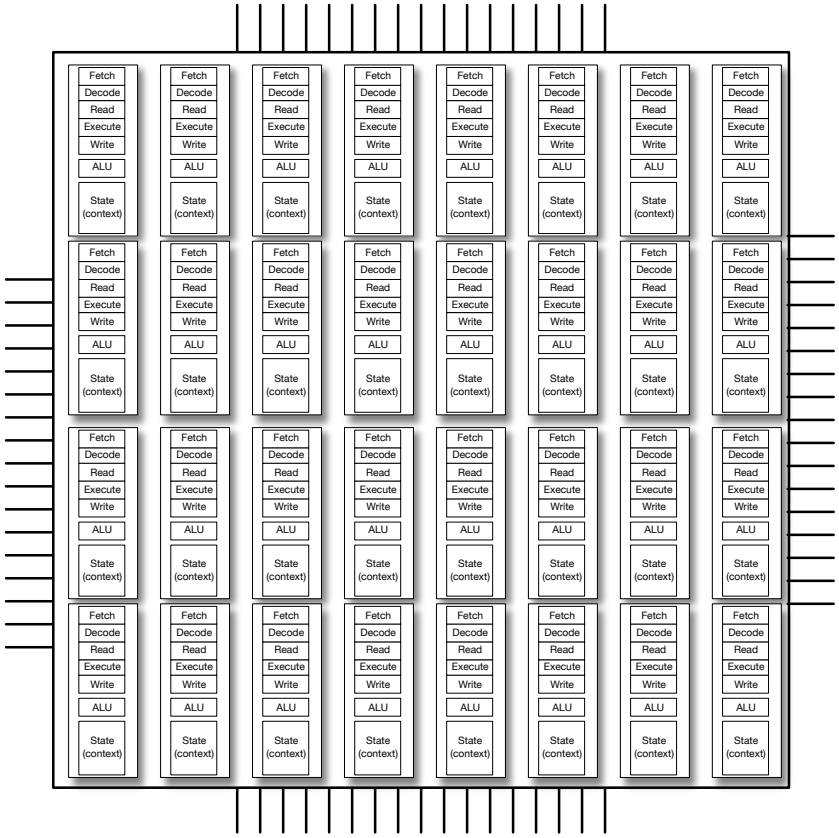
For independent parts of the scene

Eight is better than two, innit?



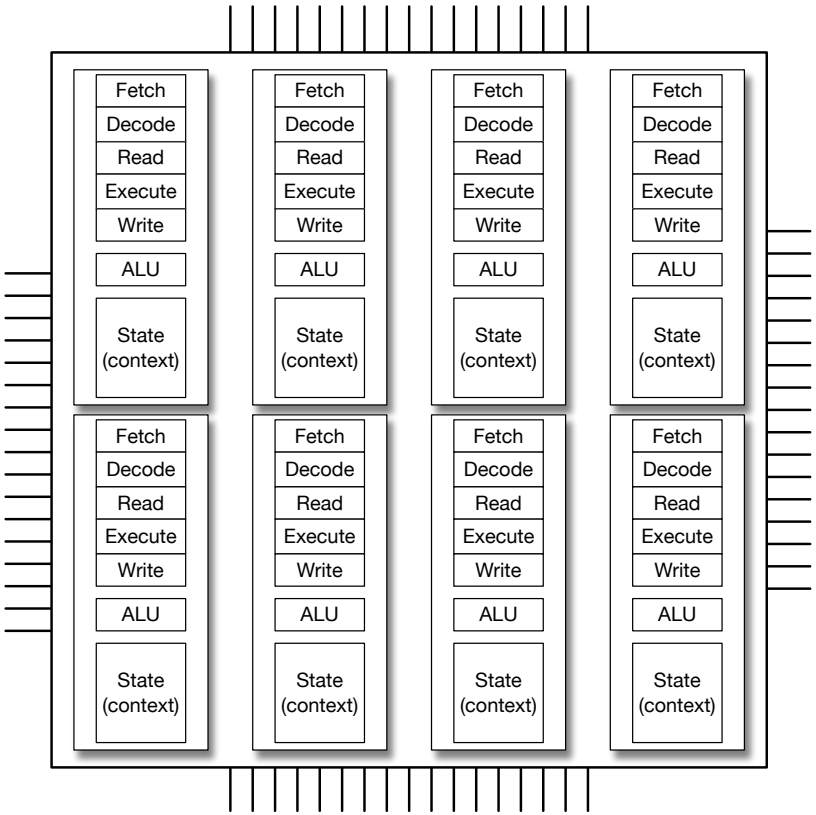


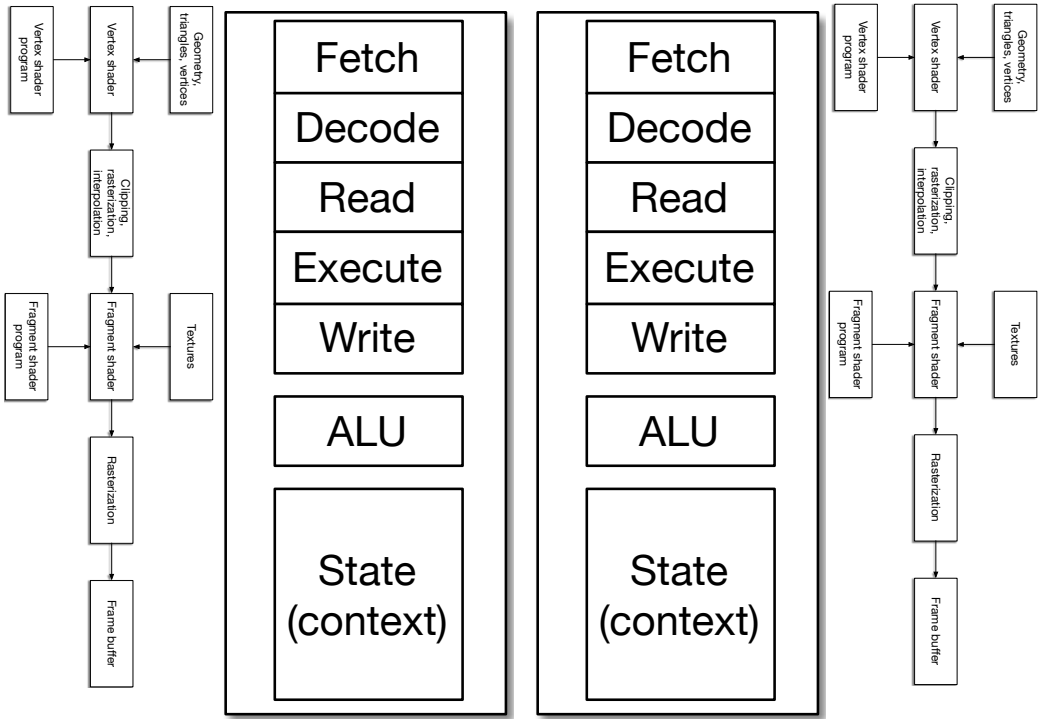
But there is just one problem



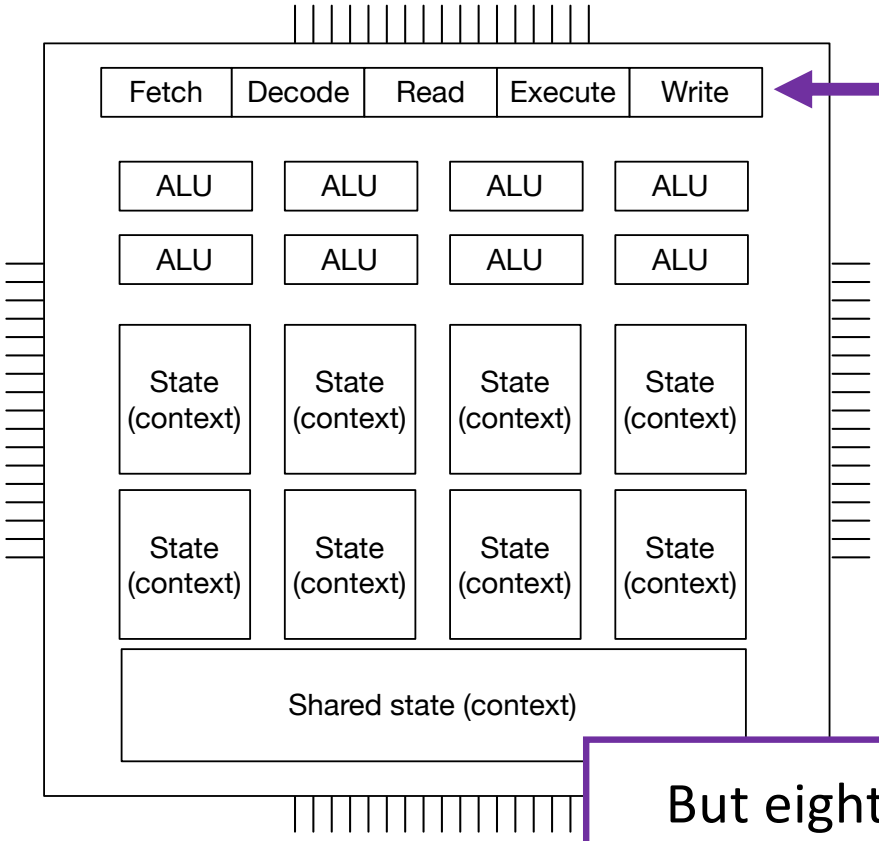
- And the problem is _____

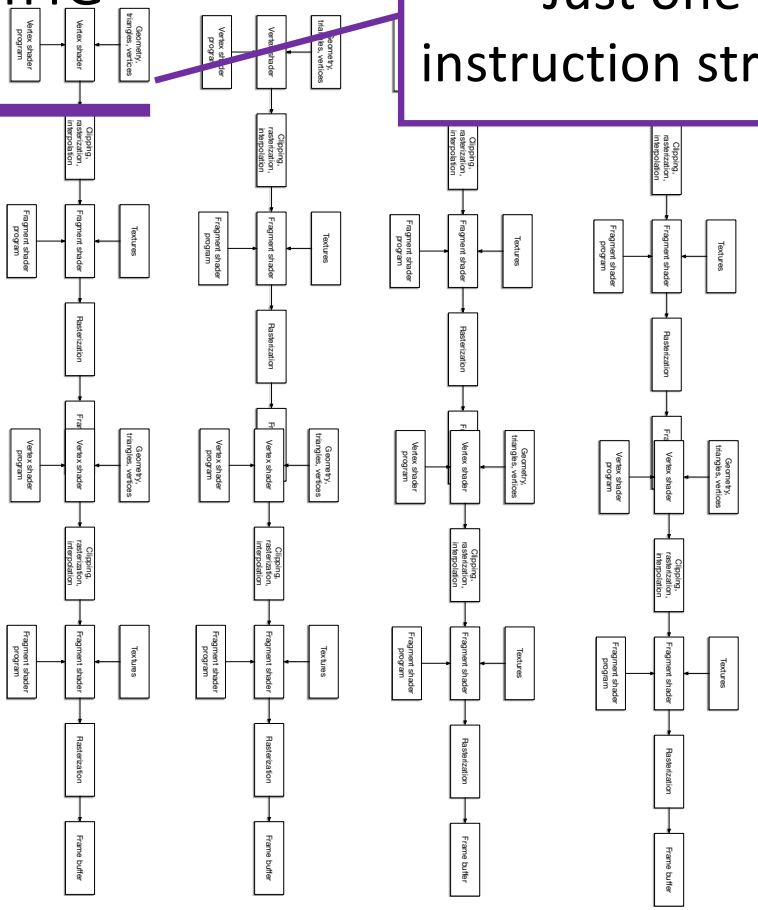
Redundant hardware





Optimizing for graphics pipeline

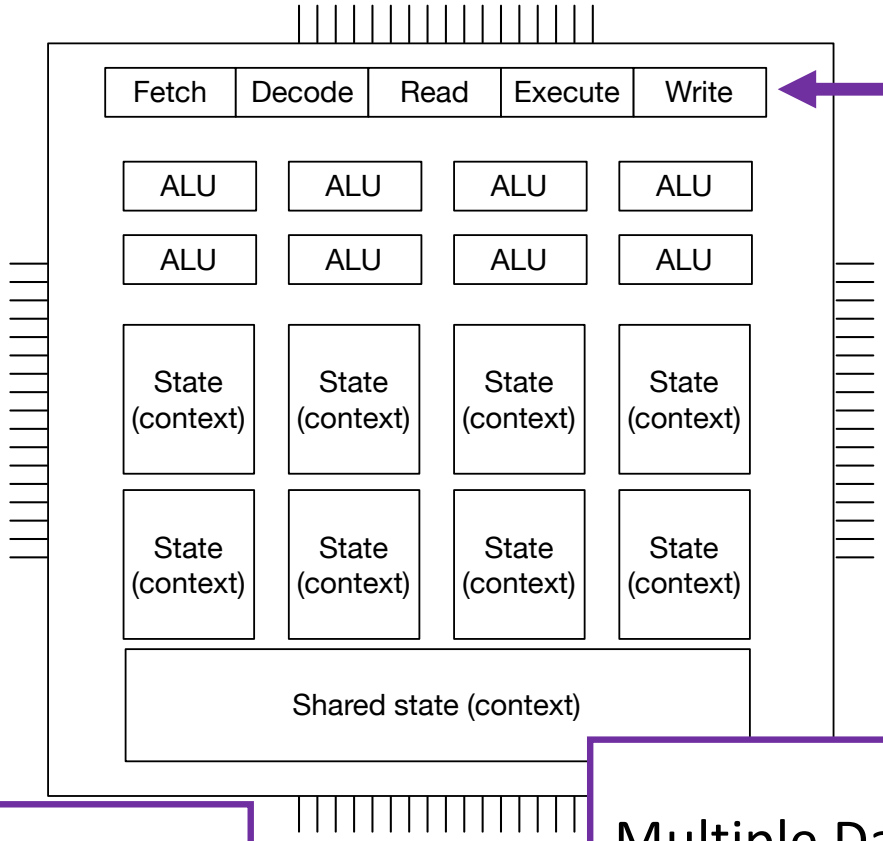


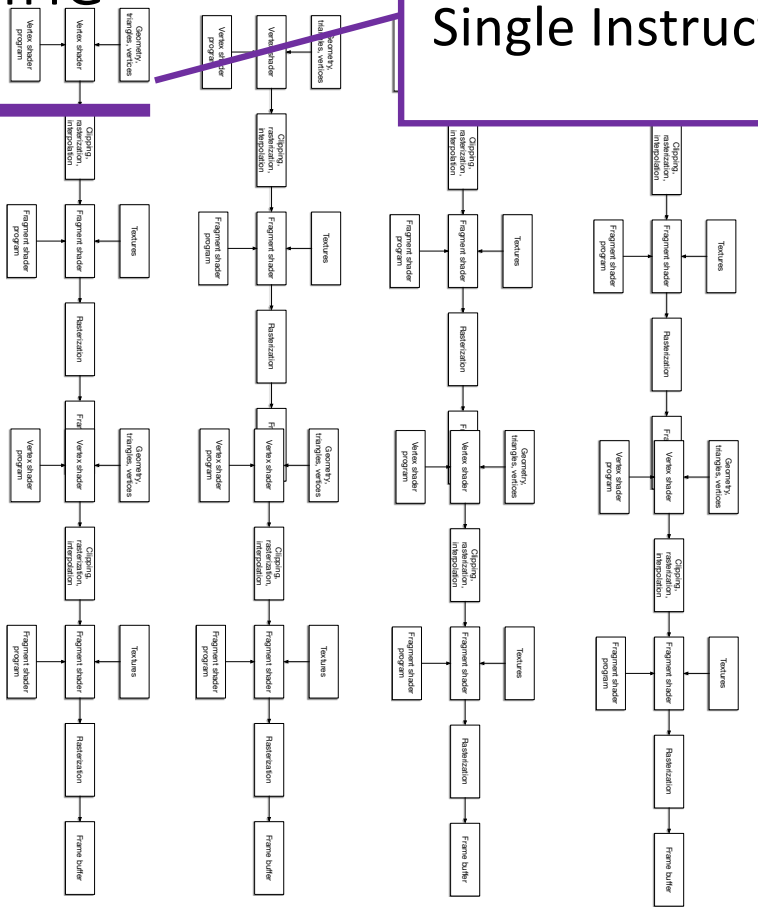


Just one instruction stream

But eight pipelines

Optimizing for graphics pipeline





Single Instruction

Multiple Data

SIMD!

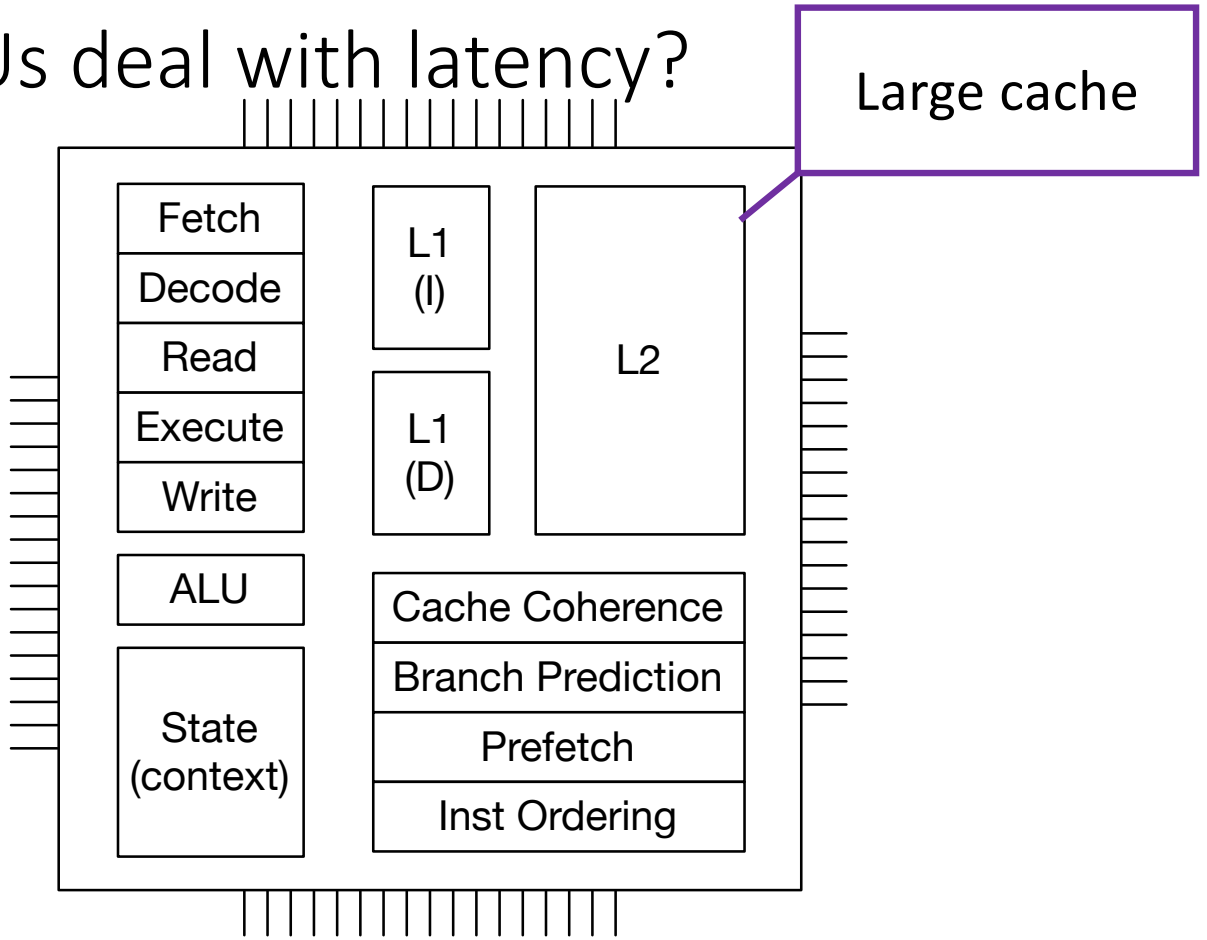
Why stop now

16 cores

128 ALUs

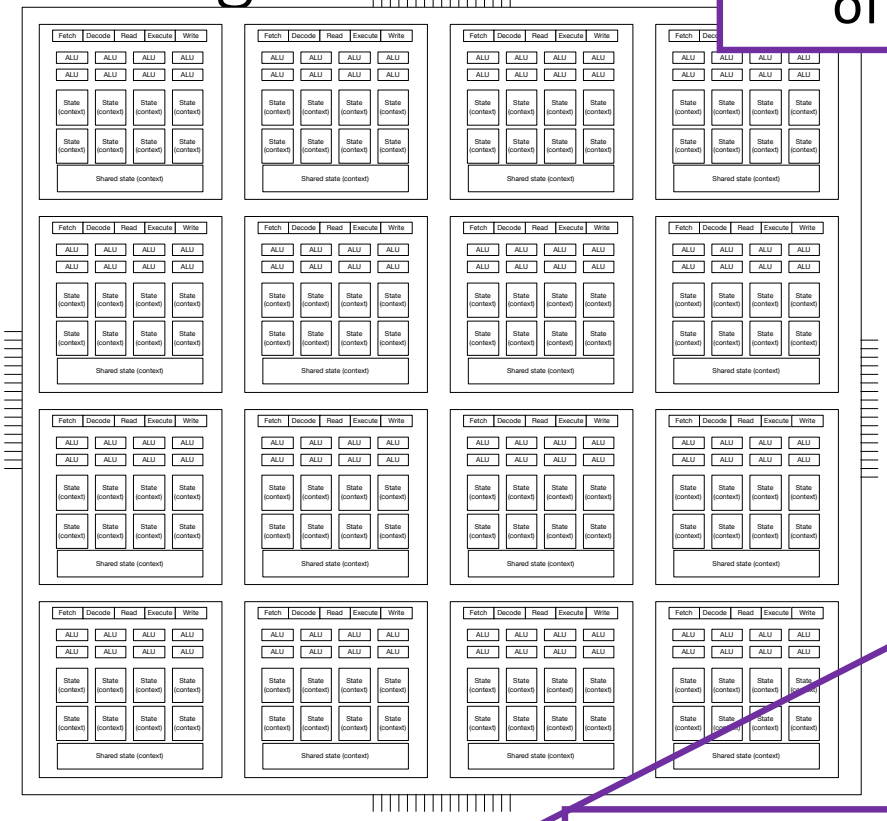
16 instruction streams

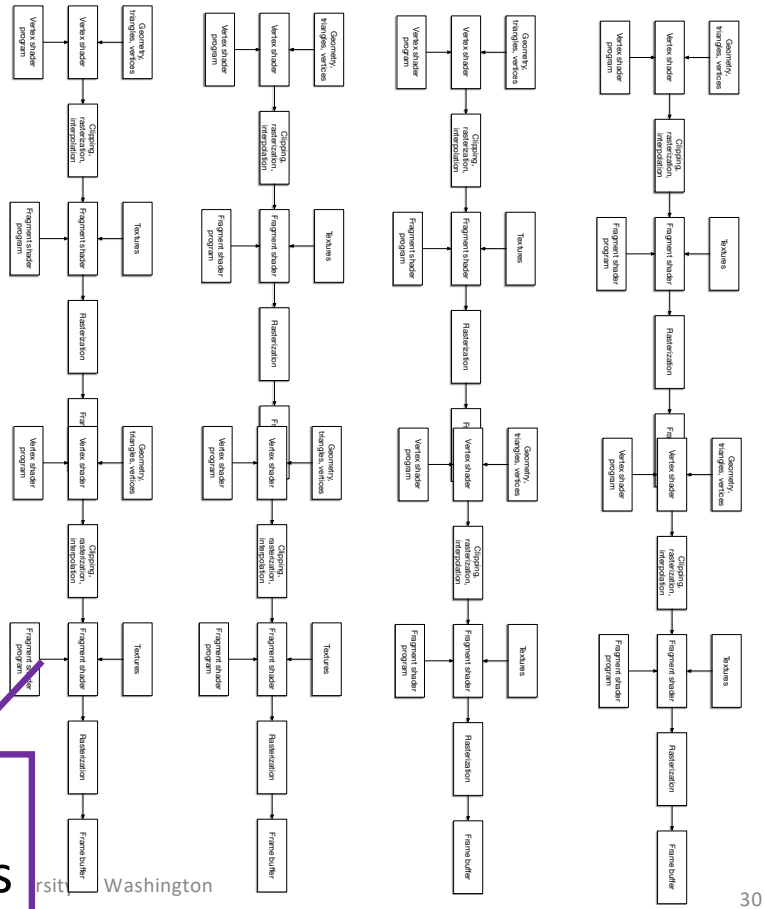
How did CPUs deal with latency?



Fragments vs Core

A real scene will have millions of fragments to process

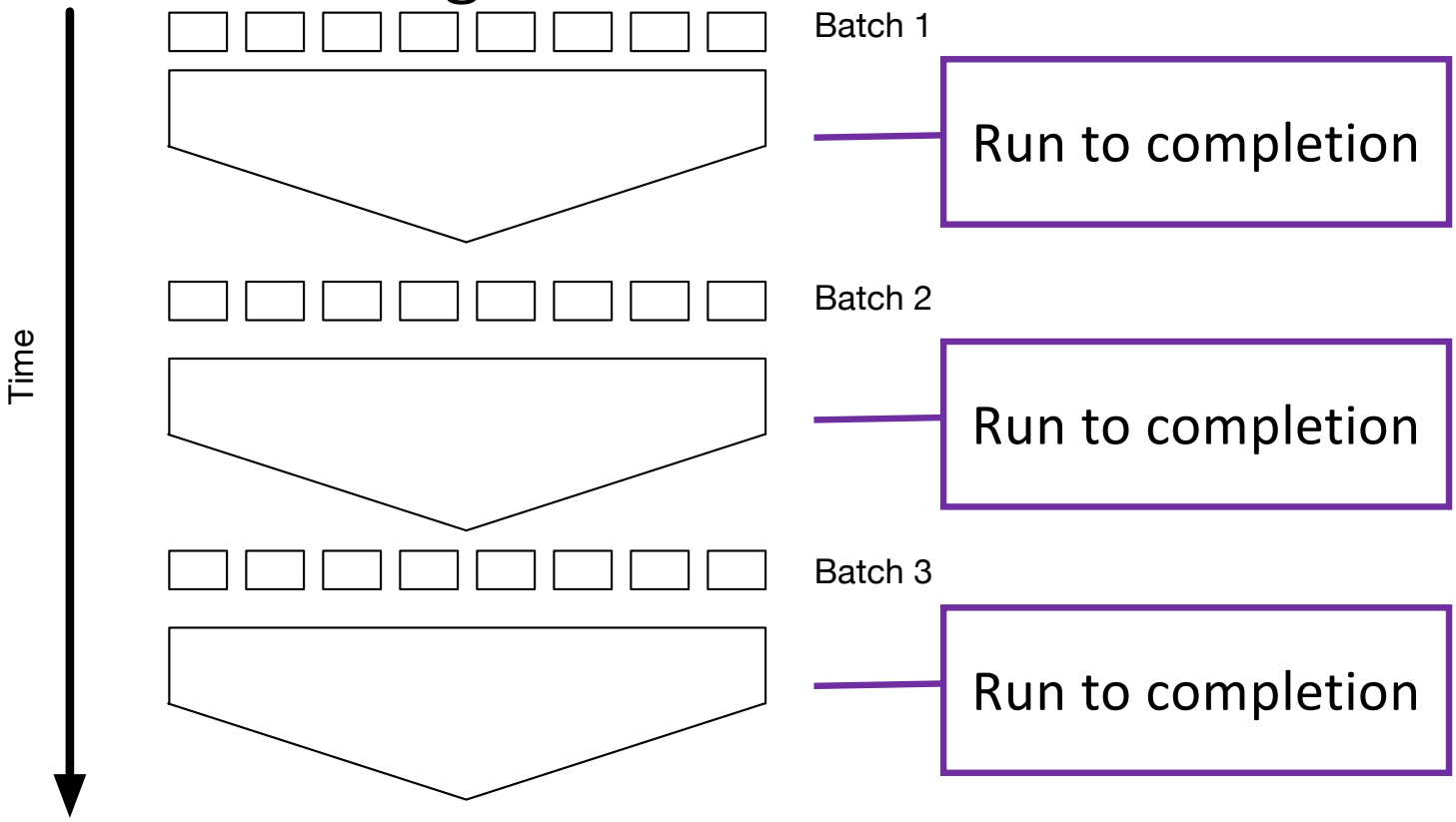




How should we divide them up?

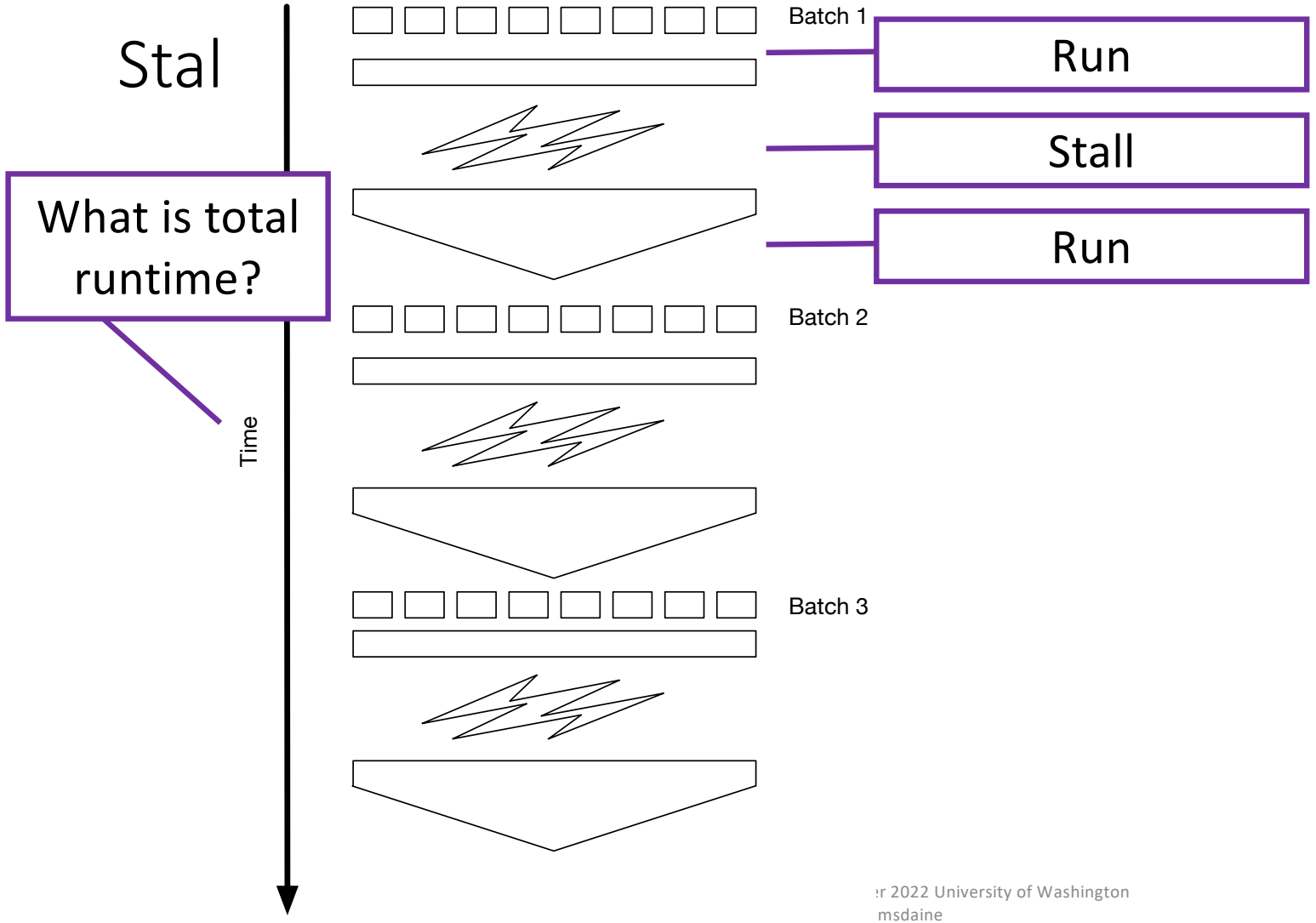
Many more fragments than cores

Scheduling

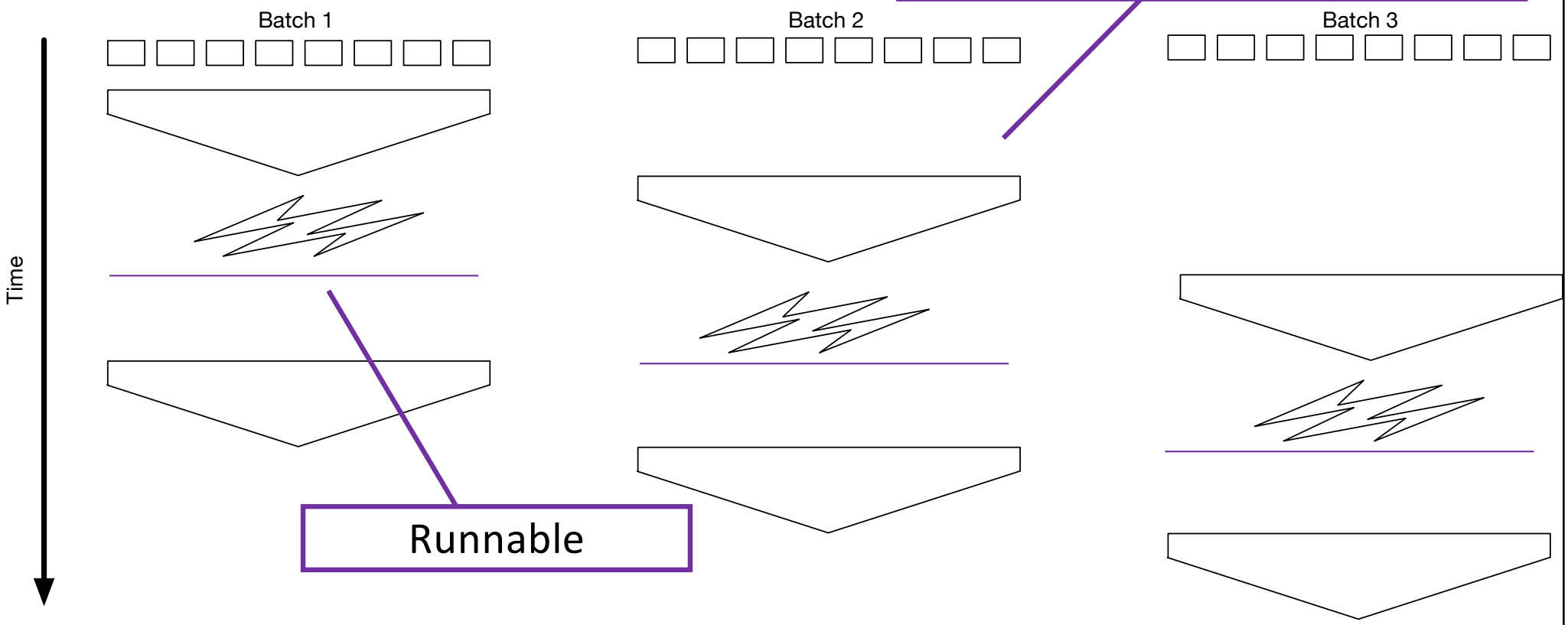

Time
Batch 1
Run to completion
Batch 2
Run to completion
Batch 3
Run to completion

Stal

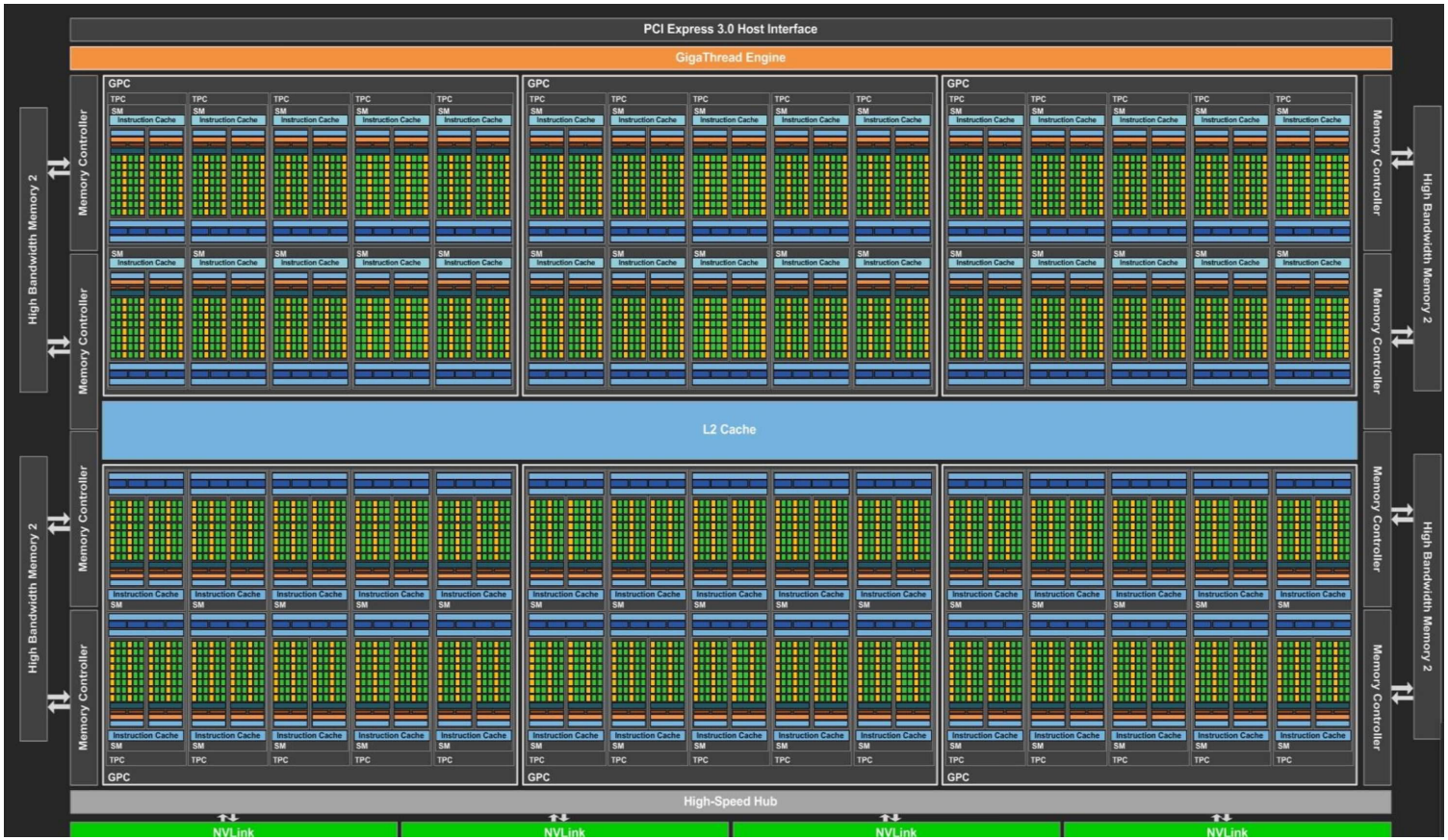
What is total runtime?



Another scheduling approach



GPU


PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
Instruction Cache
Memory Controller
High Bandwidth Memory 2
L2 Cache
High-Speed Hub
NVLink

Fermi

Streaming Multiprocessor (SM) (16)

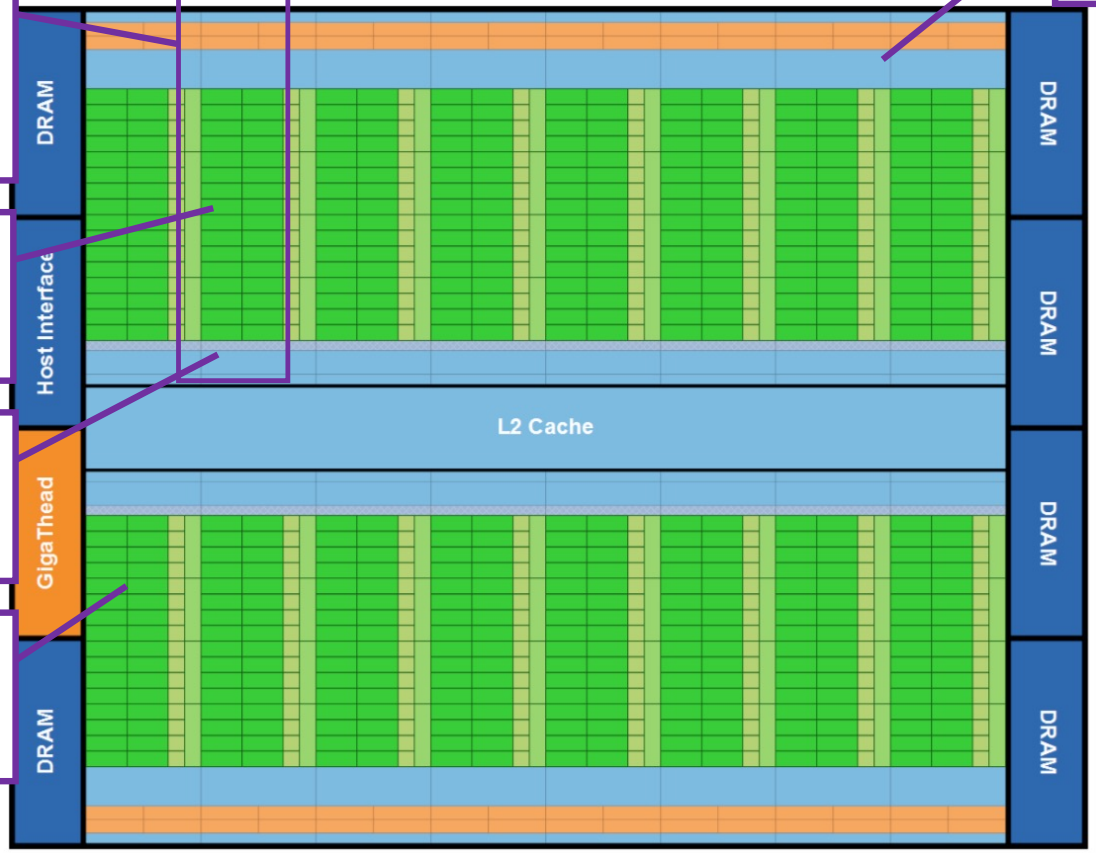
32 cores per SM

Register file and L1 cache

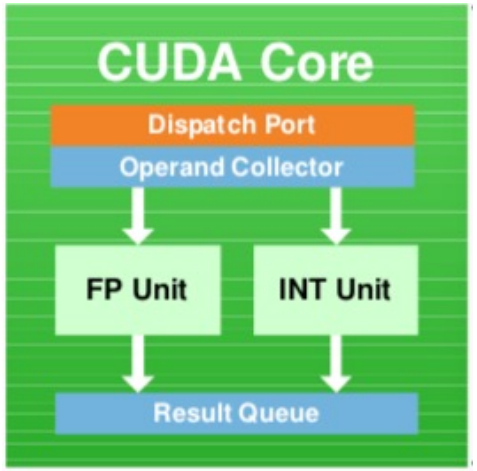
Global scheduler

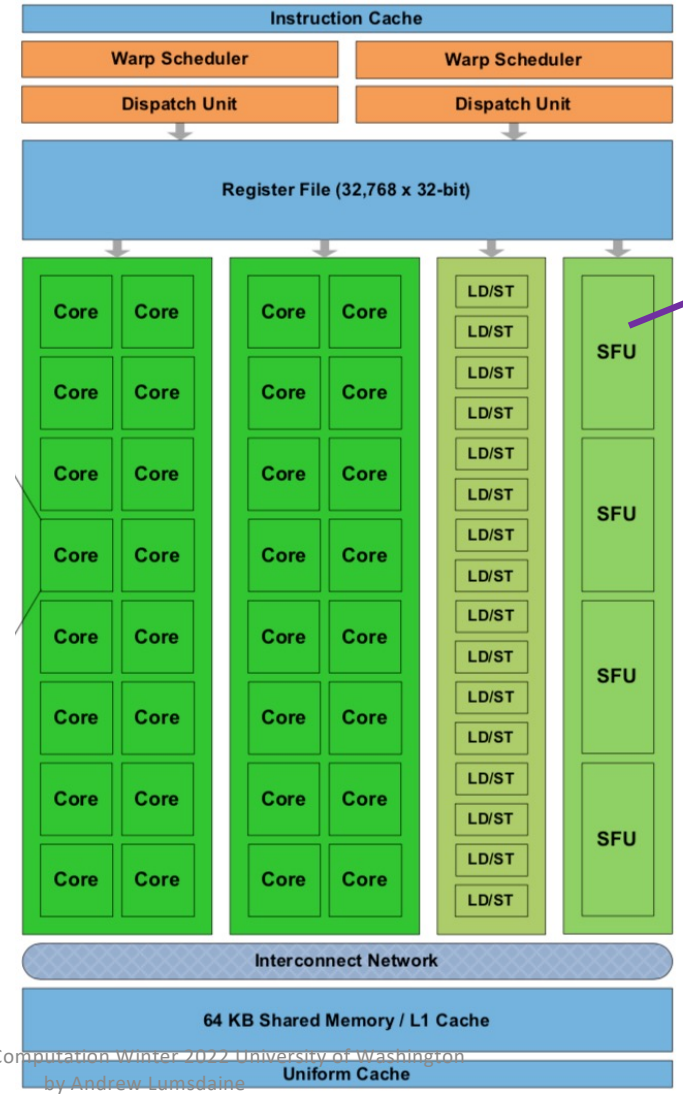
Scheduler and Dispatch

Core executes one instr/clock/thread



Fermi SM and Core





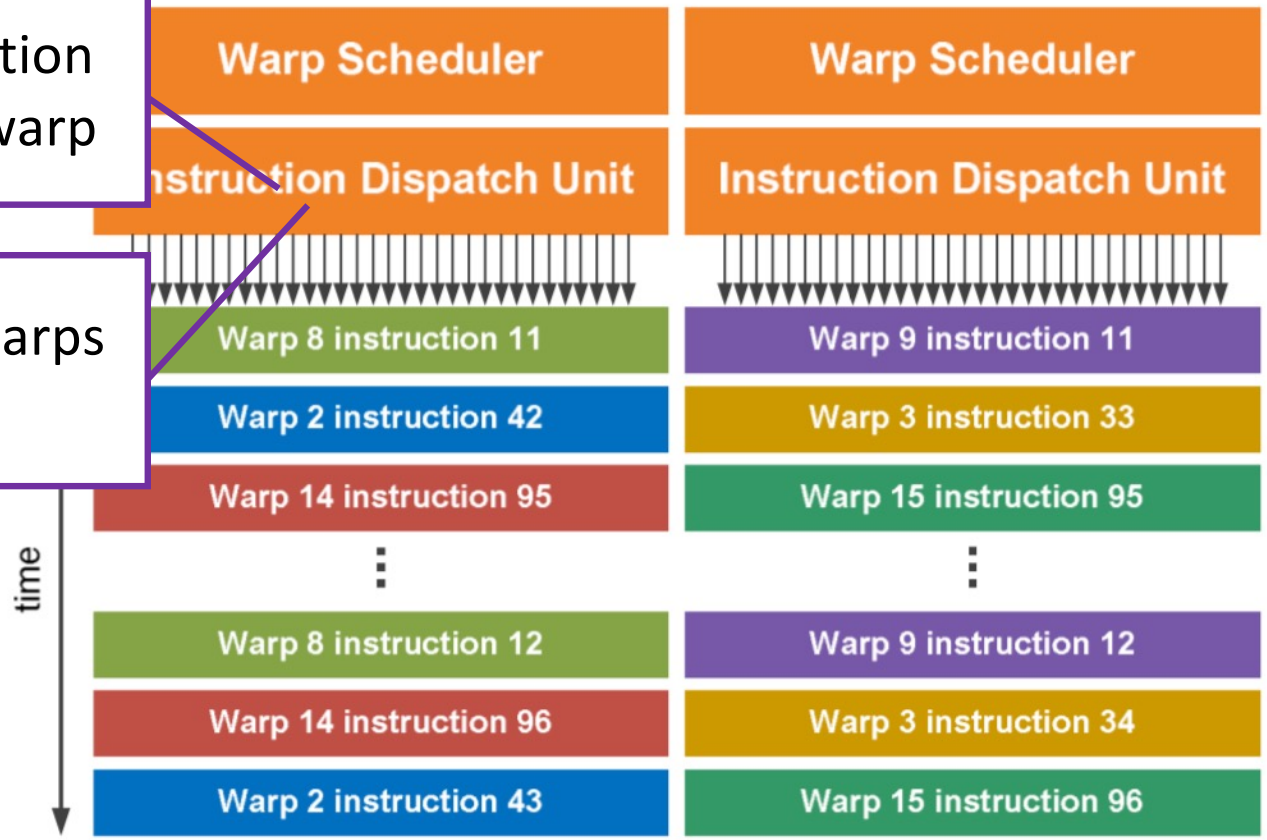
Special Function Unit

Dual Warp Scheduler

$$16 \times 32 \times 48 = 24576$$

One instruction from each warp

Up to 48* warps per SM



Fore what are GPUs optimized?

- GPUs are optimized for _____

Doing Math with Graphics Code

```
uniform sampler2D Radiance;

void main(){
  vec2 x  = gl_TexCoord[0].st * RadianceSize;
  vec2 x0 = floor(x/MicroimageSize) * MicroimageSize;
  vec2 xm = x0 + 0.5 * MicroimageSize;
  vec2 y0 = xm + (xm - x)*b_a;

  vec4 pixel = vec4(0.0, 0.0, 0.0, 1.0);
  float total_weight = 0.0;
  for (int i = -Aperture + Offset.x; i <= Aperture + Offset.x; ++i) {
    for (int j = -Aperture + Offset.y; j <= Aperture + Offset.y; ++j) {
      vec2 xmij = xm + vec2(float(i), float(j)) * MicroimageSize;
      vec2 delta = (xmij - x)*b_a;
      vec2 yij = xmij + delta;

      if (   (abs(delta.x) < gamma*0.5*MicroimageSize.x)
          && (abs(delta.y) < gamma*0.5*MicroimageSize.y)) {
        vec4 ray = texture2D(Radiance, yij / RadianceSize);
        pixel += ray;
        total_weight += 1.0;
      }
    }
  }

  gl_FragColor = pixel / total_weight;
}
```

Maybe there is something to it

Compute = Drawing, Kernel = Shader

```
float yval=y_old[i];
float xval=x[i];
y_new[i]=yval+alpha*xval;
```

```
float saxpy (
    float2 coords : TEXCOORD0,
    uniform sampler2D textureY,
    uniform sampler2D textureX,
    uniform float alpha ) : COLOR
{
    float result;
    float y = tex2D(textureY,coords);
    float x = tex2D(textureX,coords);
    result = y + alpha * x;
    return result;
}
```

```
for (int i=0; i<N; i++)
    dataY[i] = dataY[i] + alpha * dataX[i];
```

Lots and lots of graphics code

```
include <stdio.h>
#include <stdlib.h>
#include <GL/glew.h>
#include <GL/glut.h>

int main(int argc, char **argv) {
    // declare texture size, the actual data will be a vector
    // of size texSize*texSize*4
    int texSize = 2;
    // create test data
    float* data = (float*)malloc(4*texSize*texSize*sizeof(float));
    float* result = (float*)malloc(4*texSize*texSize*sizeof(float));
    for (int i=0; i<texSize*texSize*4; i++)
        data[i] = i+1.0;
    // set up glut to get valid GL context and
    // get extension entry points
    glutInit (&argc, argv);
    glutCreateWindow("TEST1");
    glewInit();
```

Lots and lots

```
// viewport transform for 1:1 pixel=texel=data mapping
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0,texSize,0.0,texSize);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glViewport(0,0,texSize,texSize);
// create FBO and bind it (that is, use offscreen render target)
GLuint fb;
glGenFramebuffersEXT(1,&fb);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT,fb);
// create texture
GLuint tex;
glGenTextures (1, &tex);
glBindTexture(GL_TEXTURE_RECTANGLE_ARB,tex);
```

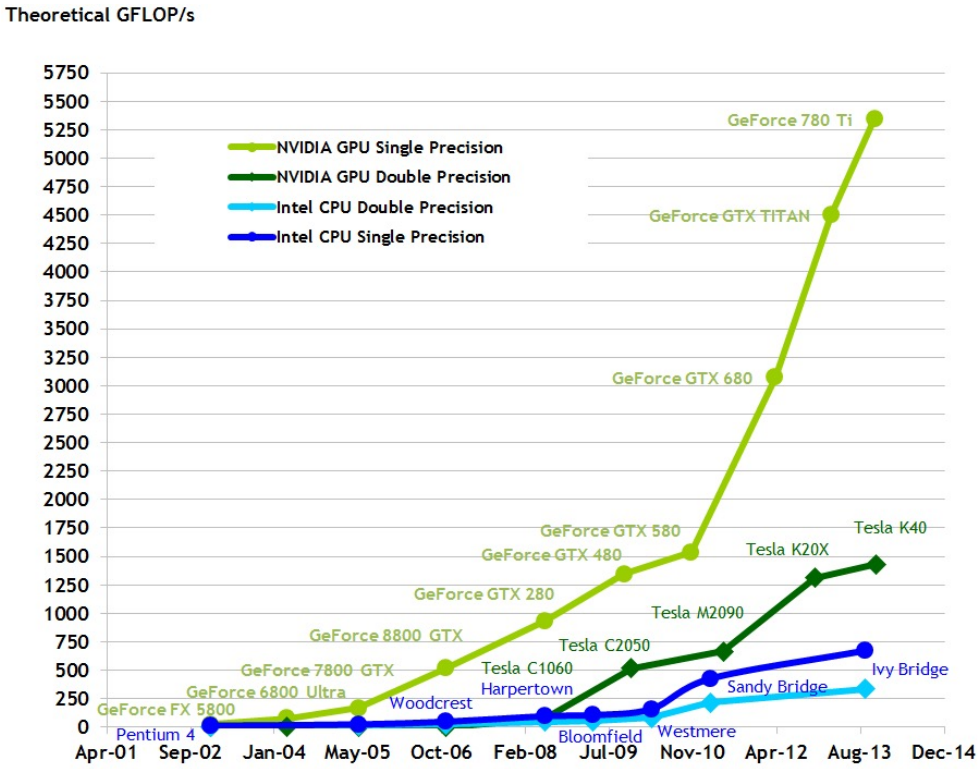
Still more

```
// set texture parameters
  glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,
                  GL_TEXTURE_MIN_FILTER, GL_NEAREST);
  glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,
                  GL_TEXTURE_MAG_FILTER, GL_NEAREST);
  glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,
                  GL_TEXTURE_WRAP_S, GL_CLAMP);
  glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,
                  GL_TEXTURE_WRAP_T, GL_CLAMP);
  // define texture with floating point format
  glTexImage2D(GL_TEXTURE_RECTANGLE_ARB,0,GL_RGBA32F_ARB,
               texSize,texSize,0,GL_RGBA,GL_FLOAT,0);
  // attach texture
  glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                            GL_COLOR_ATTACHMENT0_EXT,
                            GL_TEXTURE_RECTANGLE_ARB,tex,0);
```

And…. Done

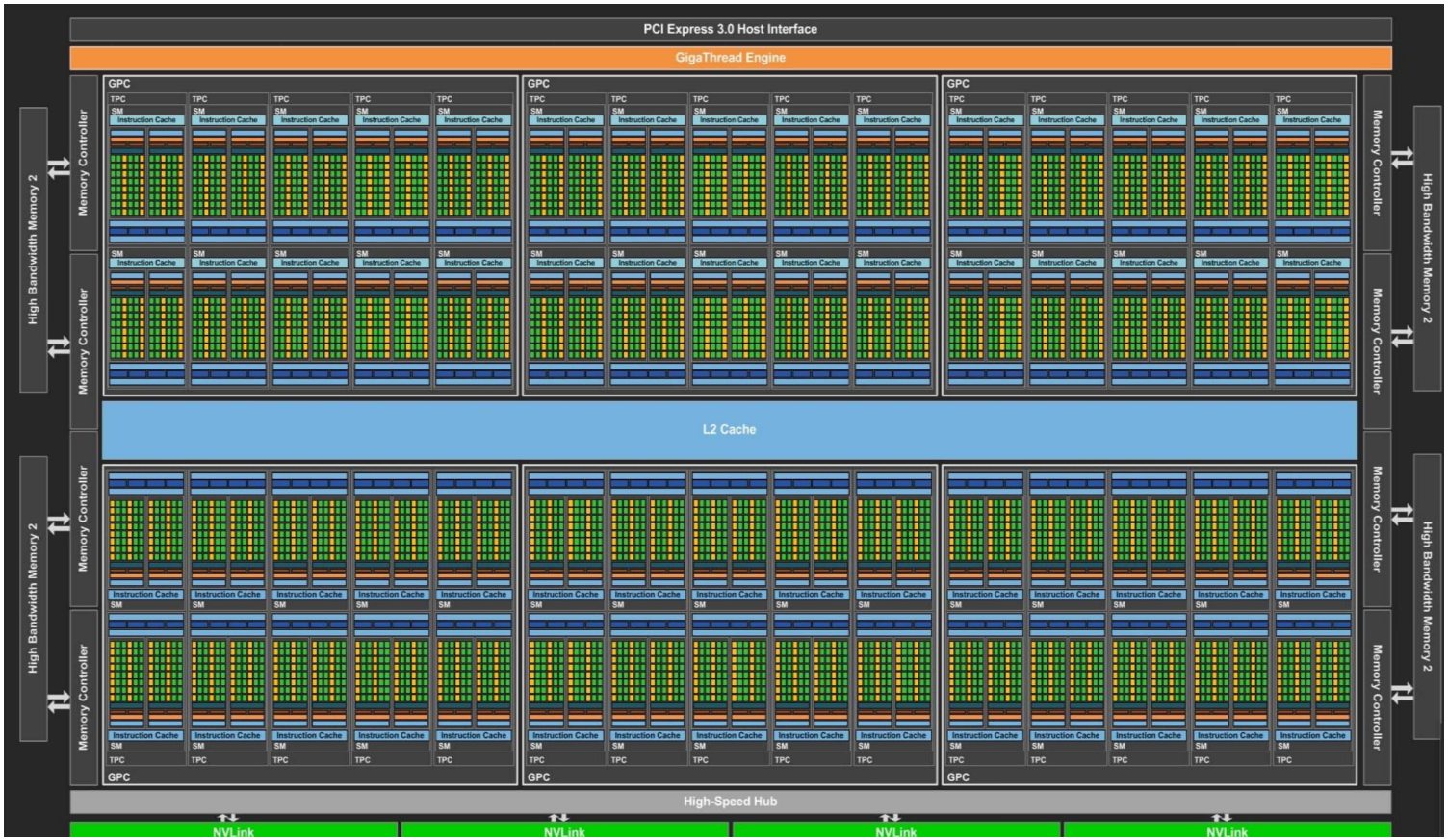
```
    // transfer data to texture
    glTexSubImage2D(GL_TEXTURE_RECTANGLE_ARB,0,0,0,texSize,texSize,
                    GL_RGBA,GL_FLOAT,data);
    // and read back
    glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
    glReadPixels(0, 0, texSize, texSize,GL_RGBA,GL_FLOAT,result);
    // print out results
    printf("Data before roundtrip:\n");
    for (int i=0; i<texSize*texSize*4; i++)
        printf("%f\n",data[i]);
    printf("Data after roundtrip:\n");
    for (int i=0; i<texSize*texSize*4; i++)
        printf("%f\n",result[i]);
    // clean up
    free(data);
    free(result);
    glDeleteFramebuffersEXT (1,&fb);
    glDeleteTextures (1,&tex);
    return 0;
}
```

Pain vs Gain



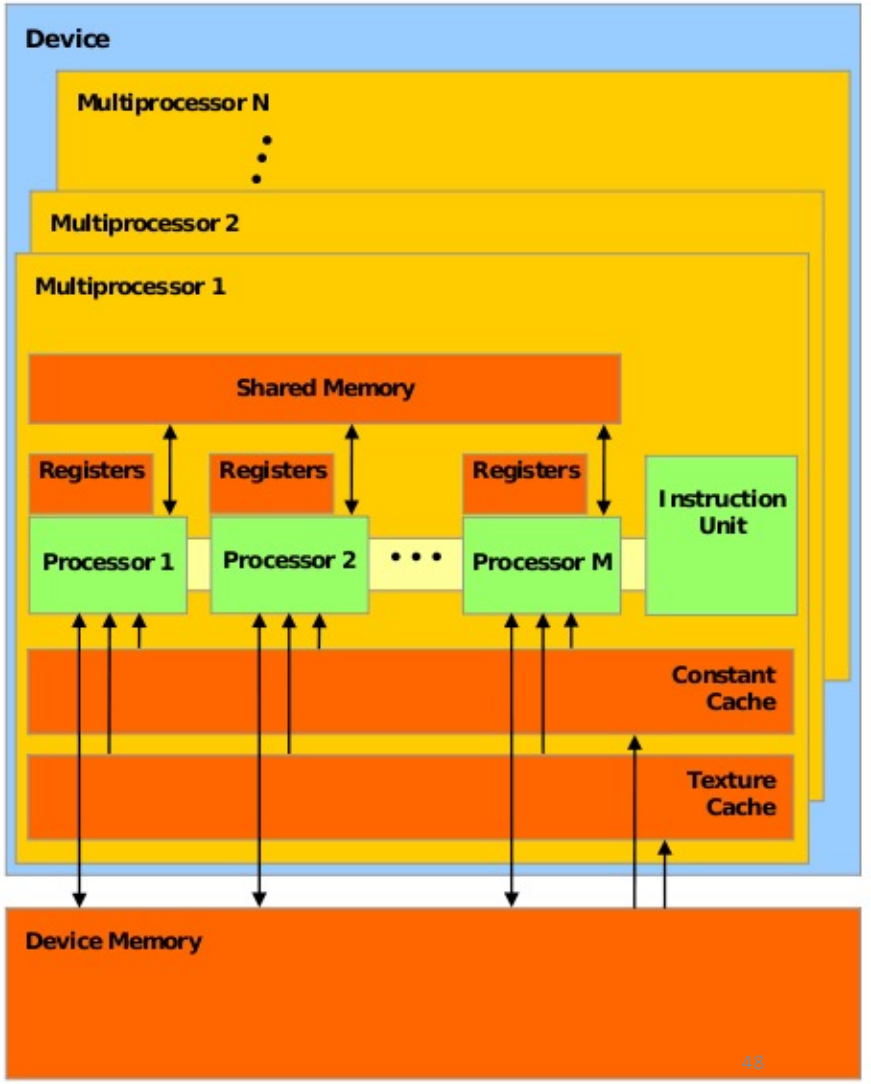
Floating-Point Operations per Second - Nvidia CUDA C Programming Guide
Version 6.5 - 24/9/2014 - copyright Nvidia Corporation 2014

GPU


PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
Instruction Cache
Memory Controller
High Bandwidth Memory 2
L2 Cache
High-Speed Hub
NVLink

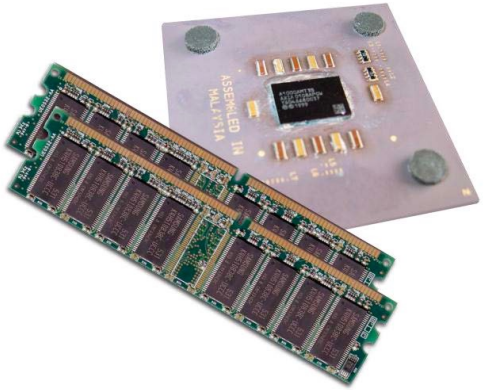
GPU architecture

- What are the important considerations re: performance?



GPU programming

- Host: CPU and its memory
  - Host memory
- Device: GPU and its memory
  - Device memory



Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

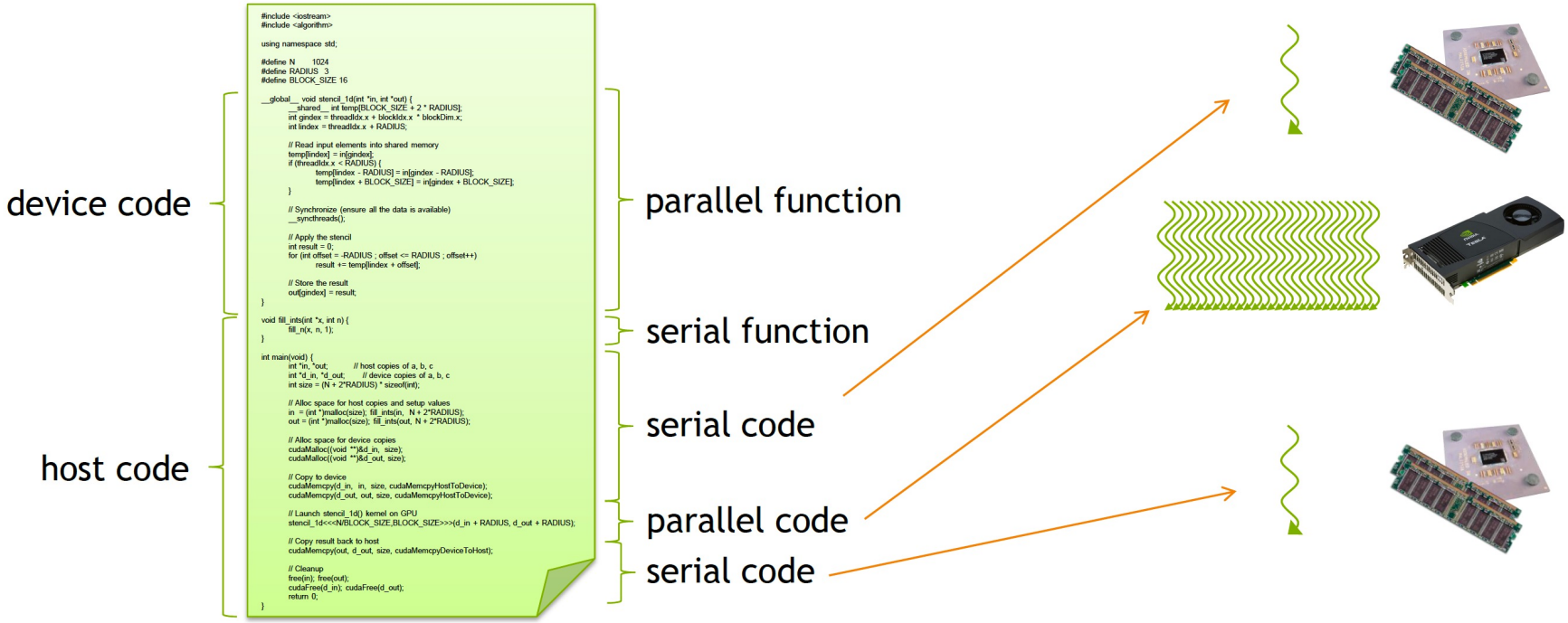
__syncthreads()

Asynchronous operation

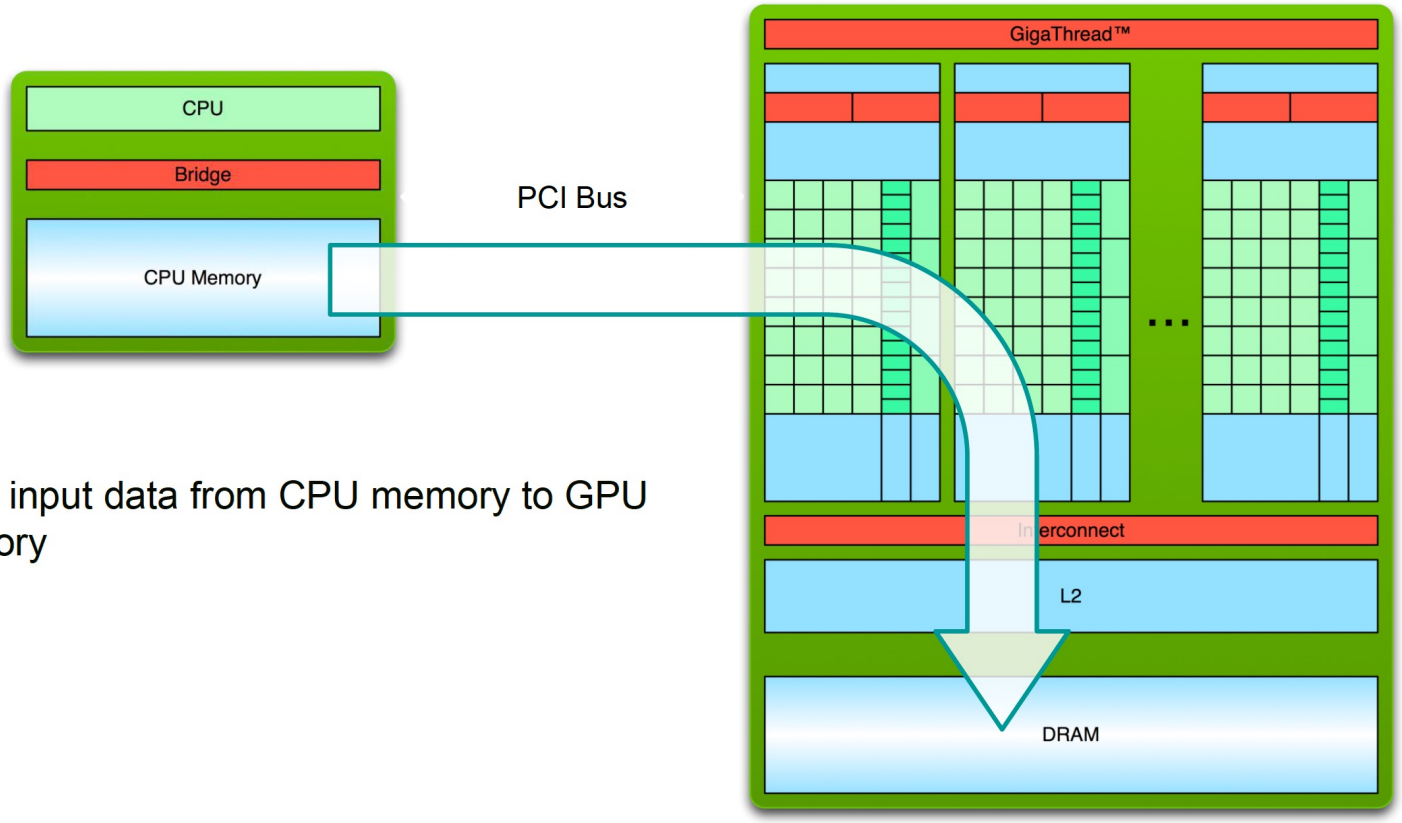
Handling errors

Managing devices

Heterogeneous computing

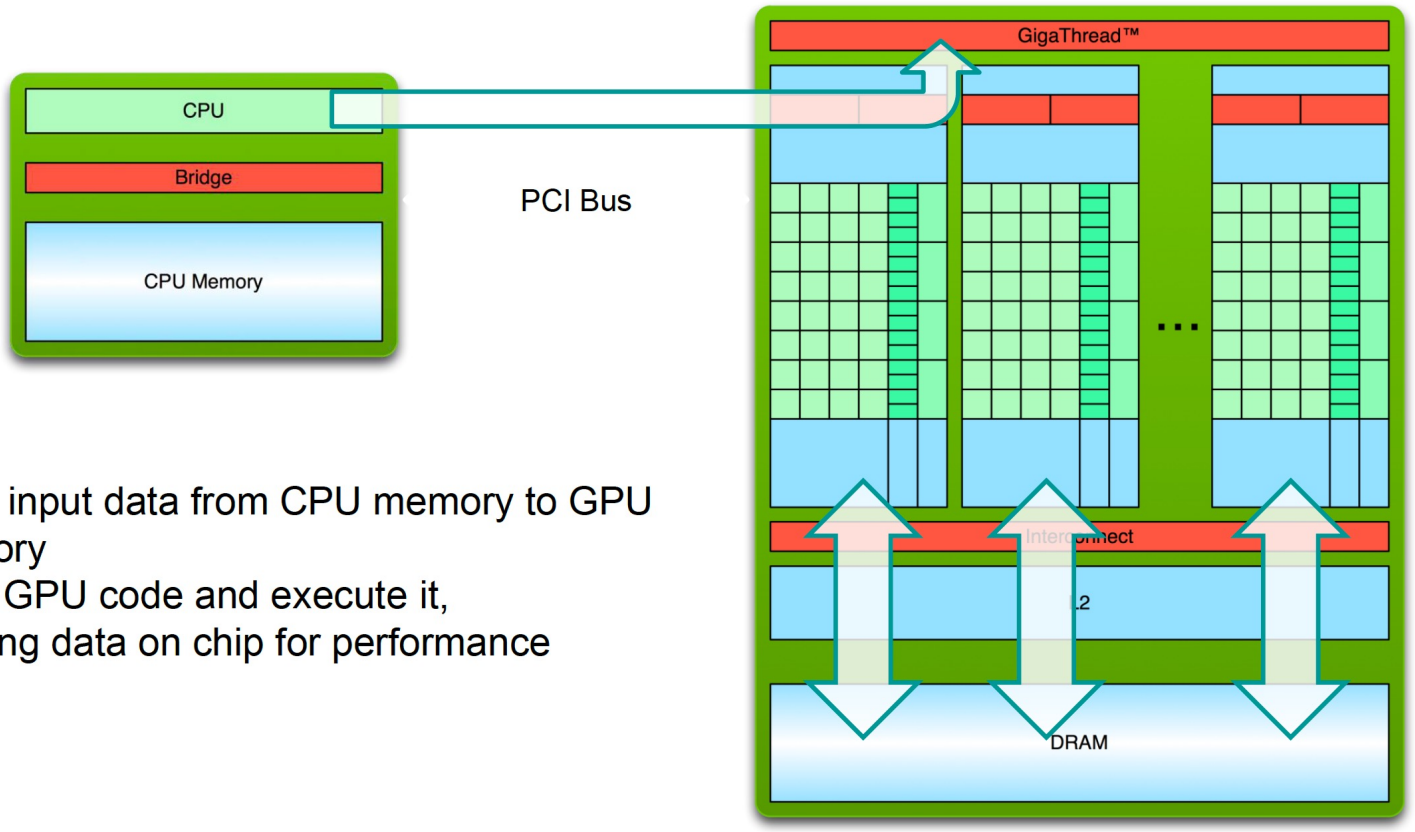


Simple processing flow



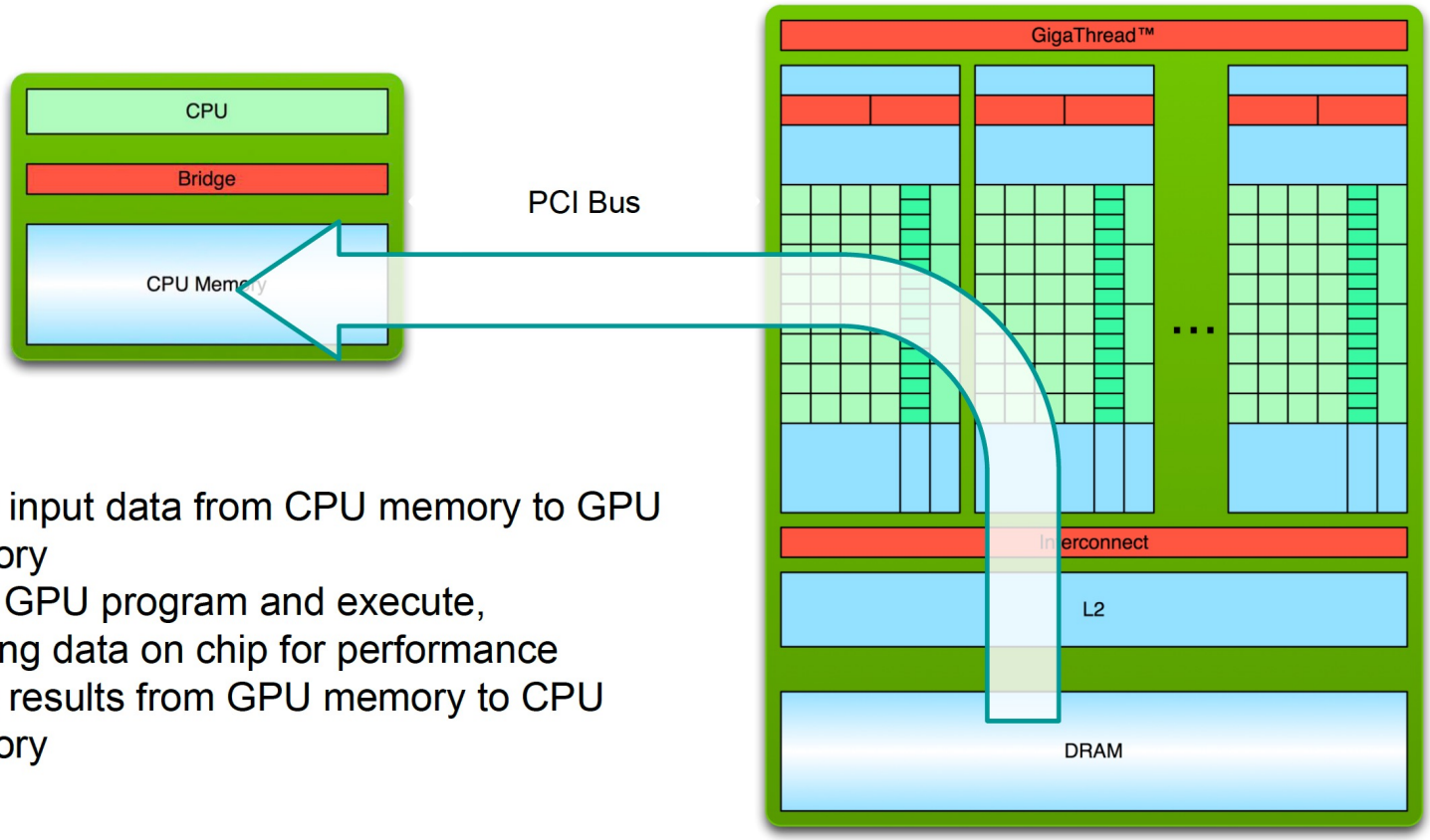
1. Copy input data from CPU memory to GPU memory

Simple processing flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU code and execute it, caching data on chip for performance

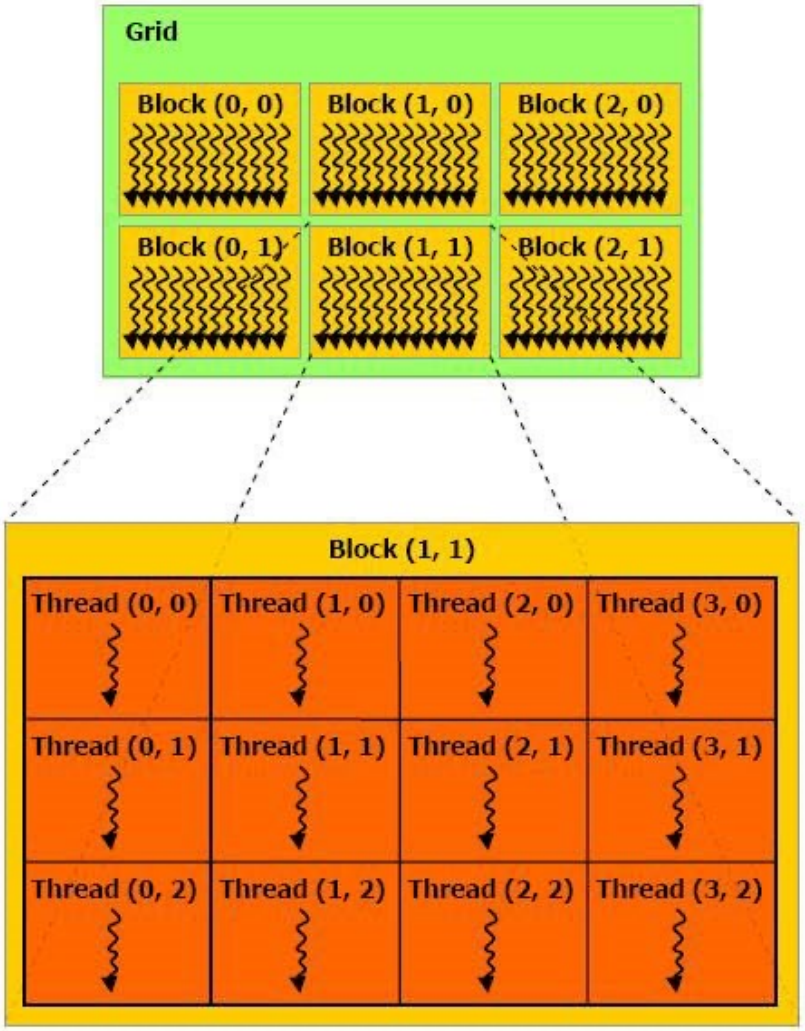
Simple processing flow



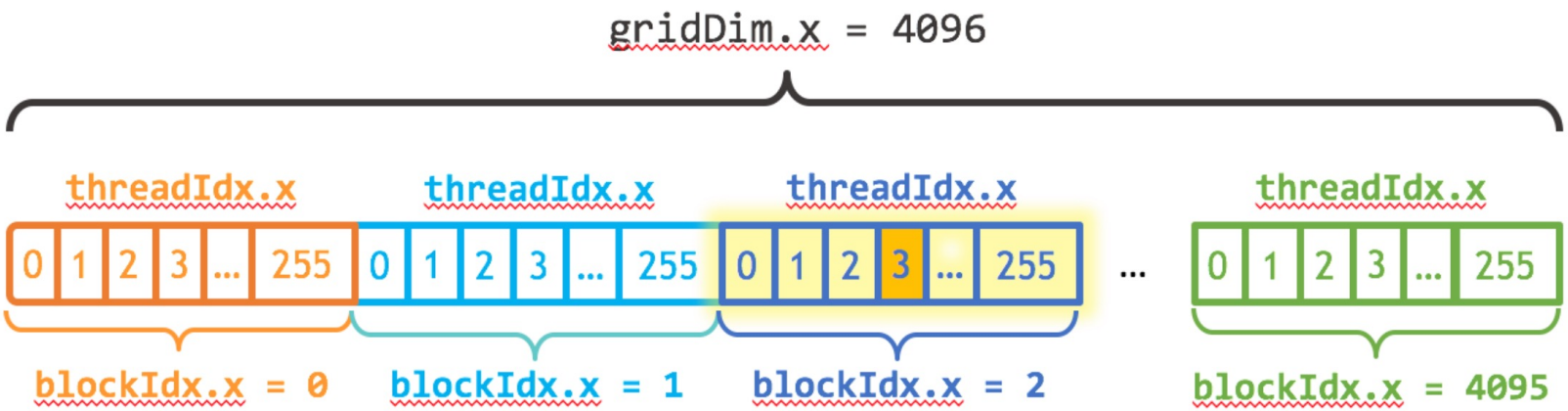
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Grids and blocks

- Thread - Distributed by the CUDA runtime (threadIdx)
- Block - A user defined group of 1 to ~512 threads (blockIdx)
- Grid - A group of one or more blocks. A grid is created for each CUDA kernel function called



Threads and blocks



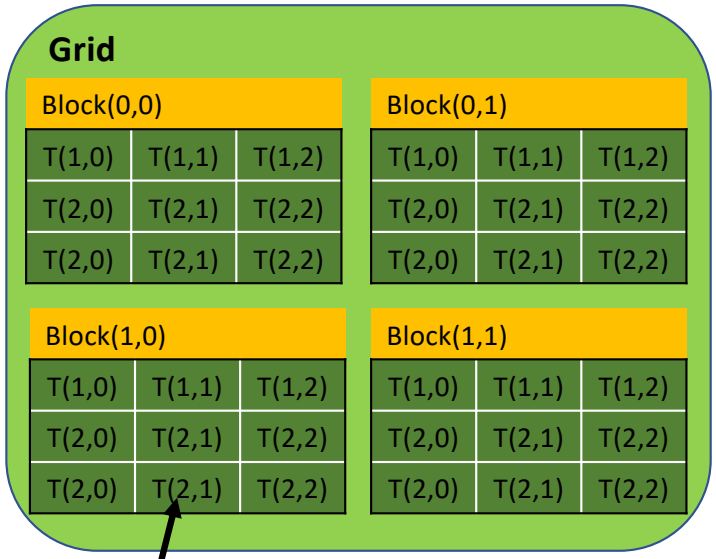

```
index = blockIdx.x  * blockDim.x + threadIdx.x

index =    (2)      *   (256)    +     (3)      = 515
```

CUDA Thread IDs to decide what data to work on

- All threads in a block execute the same kernel program
- Programmer declares thread blocks:
  - Block with up to *max* number concurrent threads
  - Block shape 1D, 2D, or 3D, Block dimensions in threads
- Threads have ***thread_id*** numbers within block
  - program uses *thread_id* to select work

```
// Kernel invocation of a 3D grid consisting of 3D blocks
__global__ void kernel() {
    // print grid and block dimensions and identifiers
    printf("Hello from thread (%d %d %d) "
           "in a block of dimension (%d %d %d) "
           "with block identifier (%d %d %d) "
           "spawned in a grid of shape (%d %d %d)\n",
           threadIdx.x, threadIdx.y, threadIdx.z,
           blockDim.x, blockDim.y, blockDim.z,
           blockIdx.x, blockIdx.y, blockIdx.z,
           gridDim.x, gridDim.y, gridDim.z);
}

// invoke the kernel
kernel <<<grid_dim, block_dim>>>();
```

```
dim3 grid_dim(2,2)
dim3 block_dim(3,3)
```

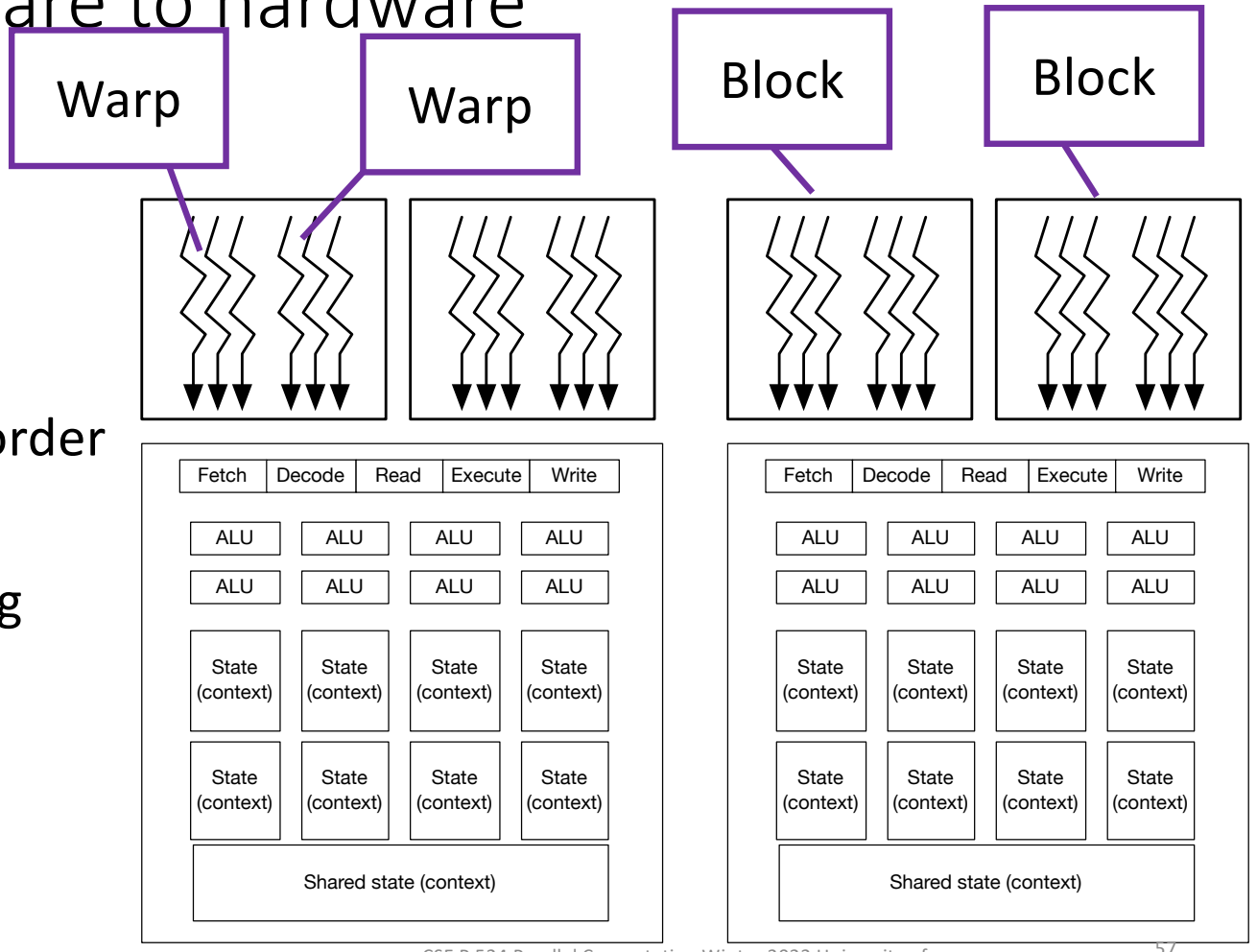



```
blockIdx.x = 1
blockIdx.y = 0
threadIdx.x = 2
threadIdx.y = 1
```

Mapping software to hardware

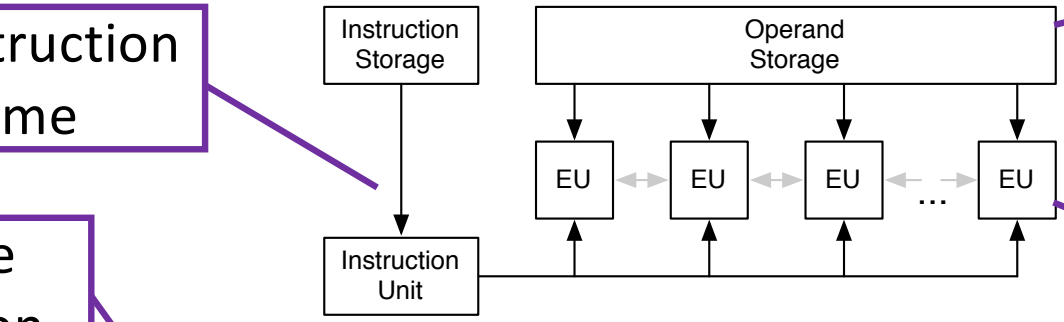
- Run in parallel or serial
- Run on the same or different SM
- Run in order or out of order

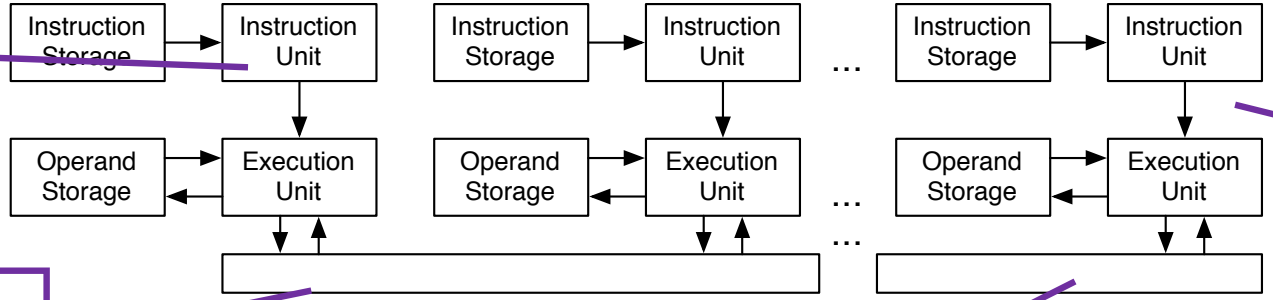
- Do not assume anything about order of blocks



SIMD and MIMD (Flynn)

- Two principal parallel computing paradigms (multiple op





Single instruction at a time

But each have their own data

Multiple instruction

All execution units execute in (c)lock step

EUs run independently (w own instrs)

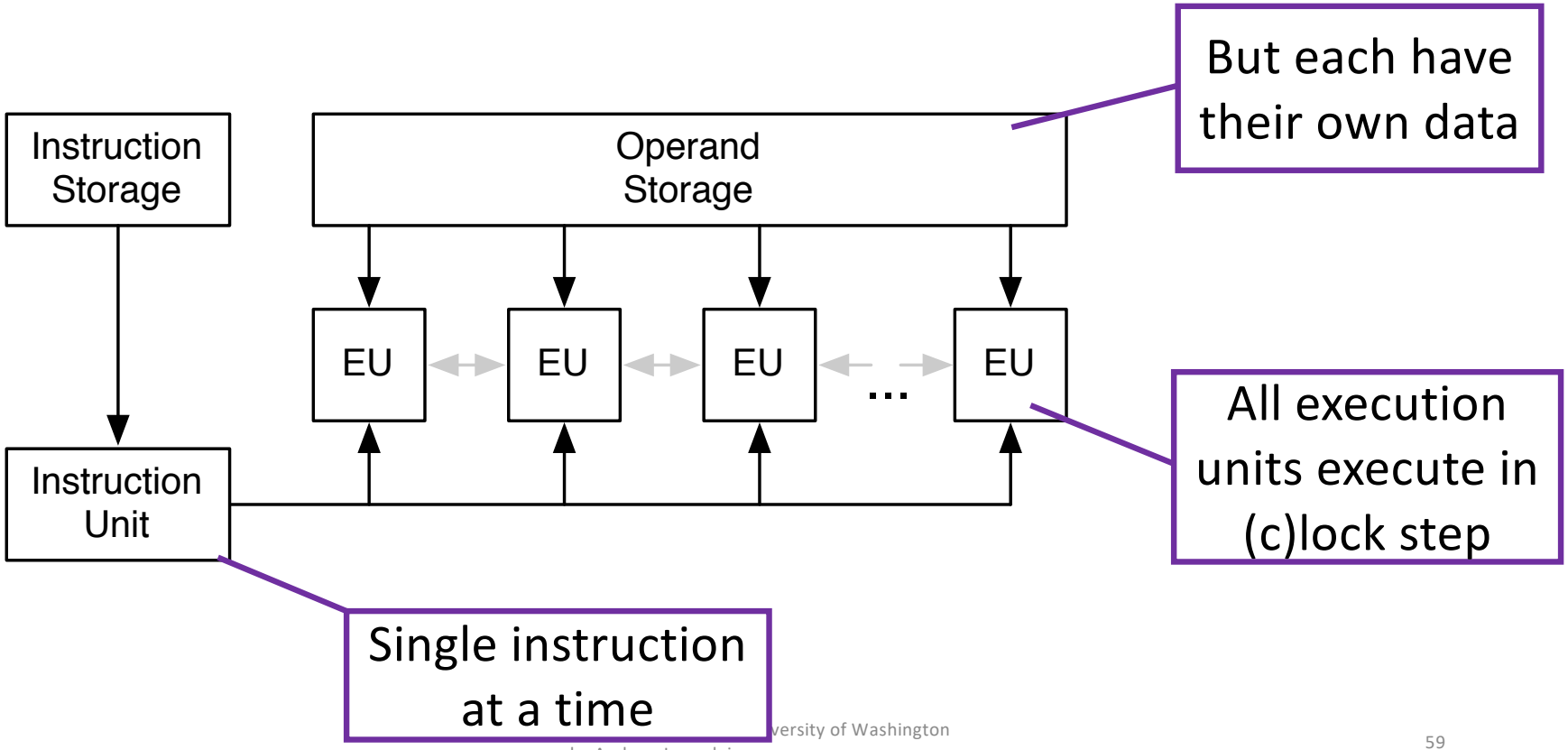
Coming up next

Shared Memory

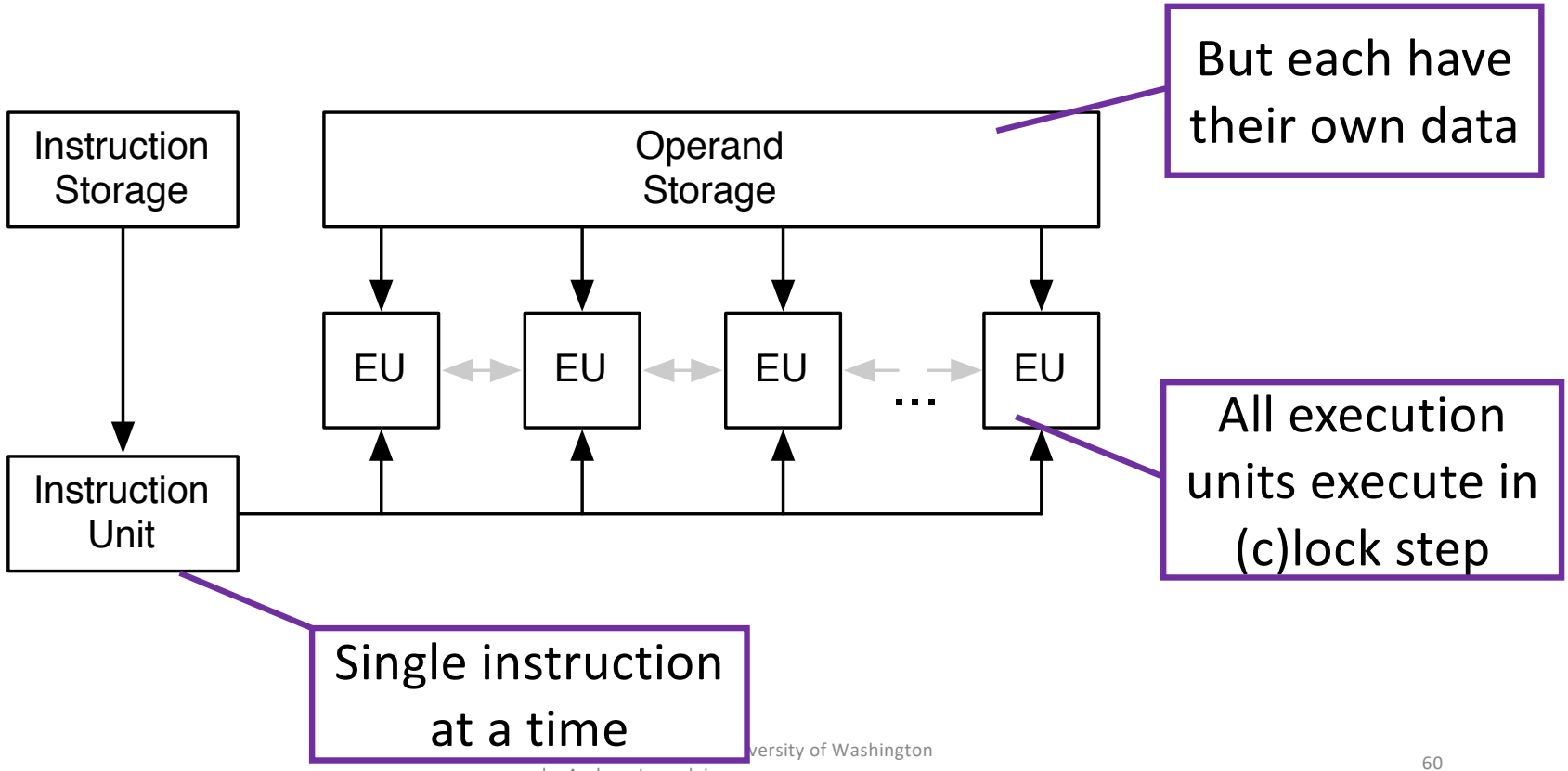
Not Shared

SIMD

- Two principal parallel computing paradigms (multiple operands)

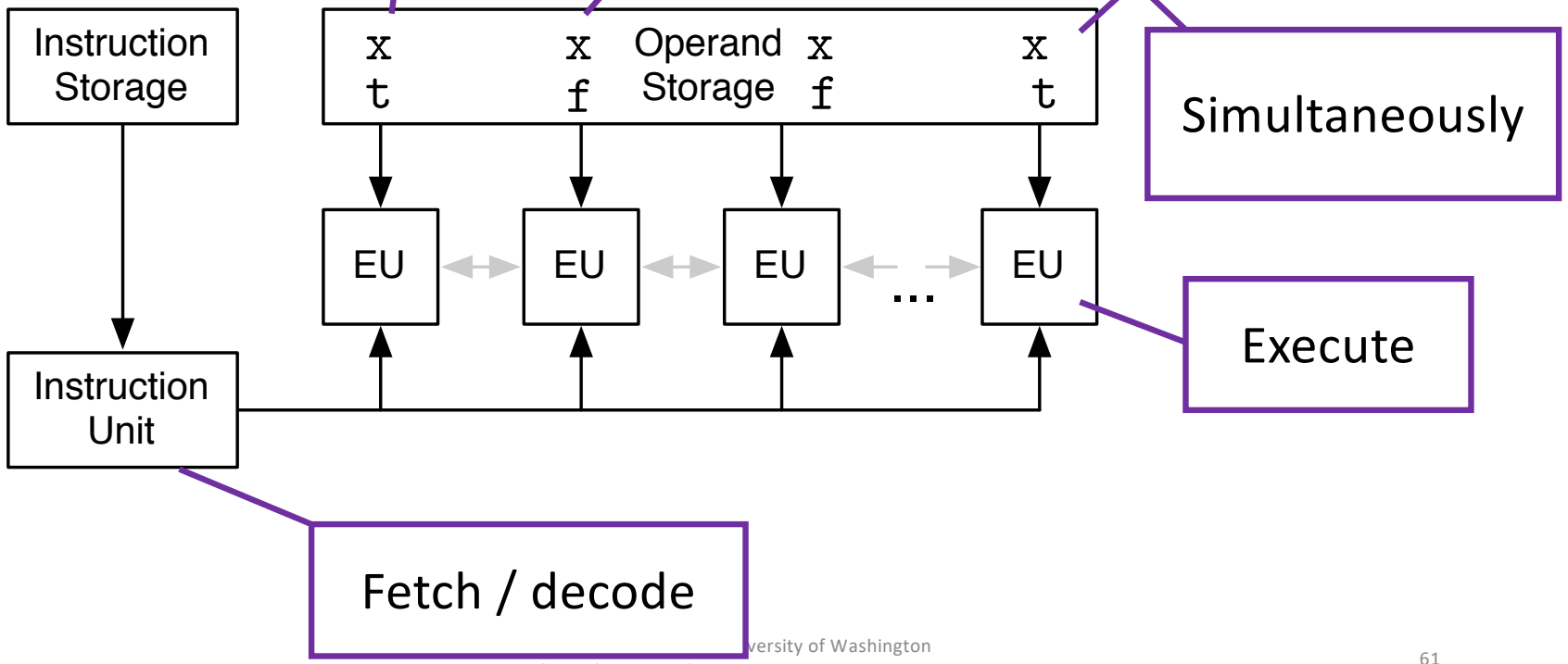


SIMD branches



SIMD branch

```
if (x < 0) {
  x = -x;
}
```



SIMD branches

```
if (x < 0) {
  x = -x;
}
```

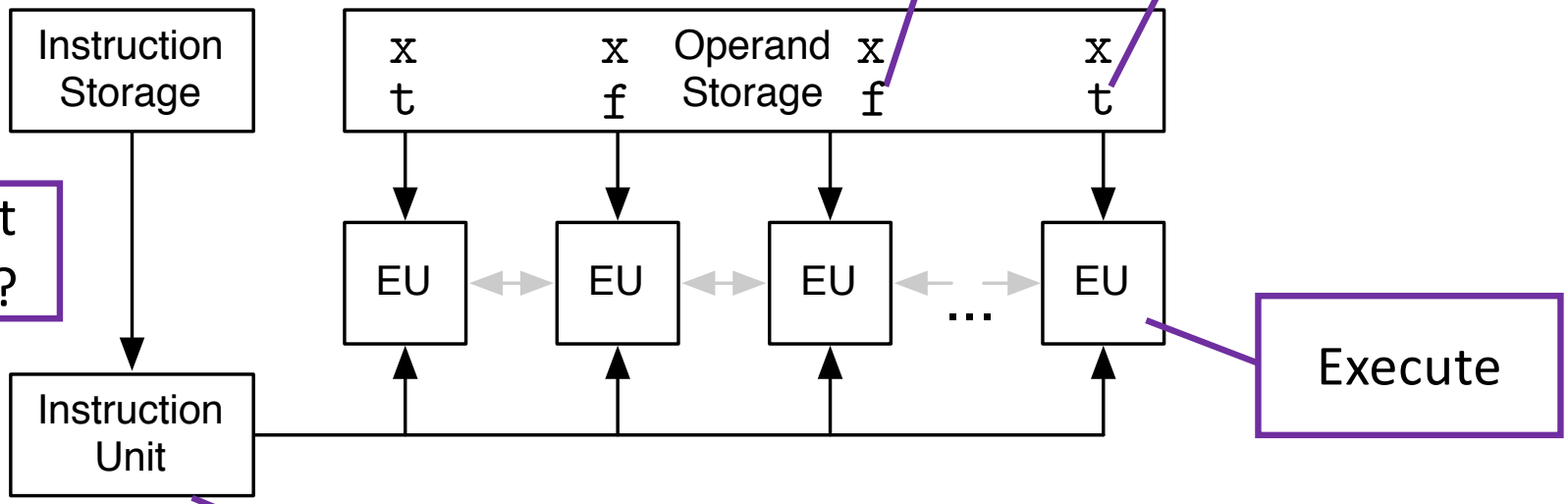
What next?

What is the next instruction?

Each have their own true/false

Each do their own evaluation

Where is it evaluated?

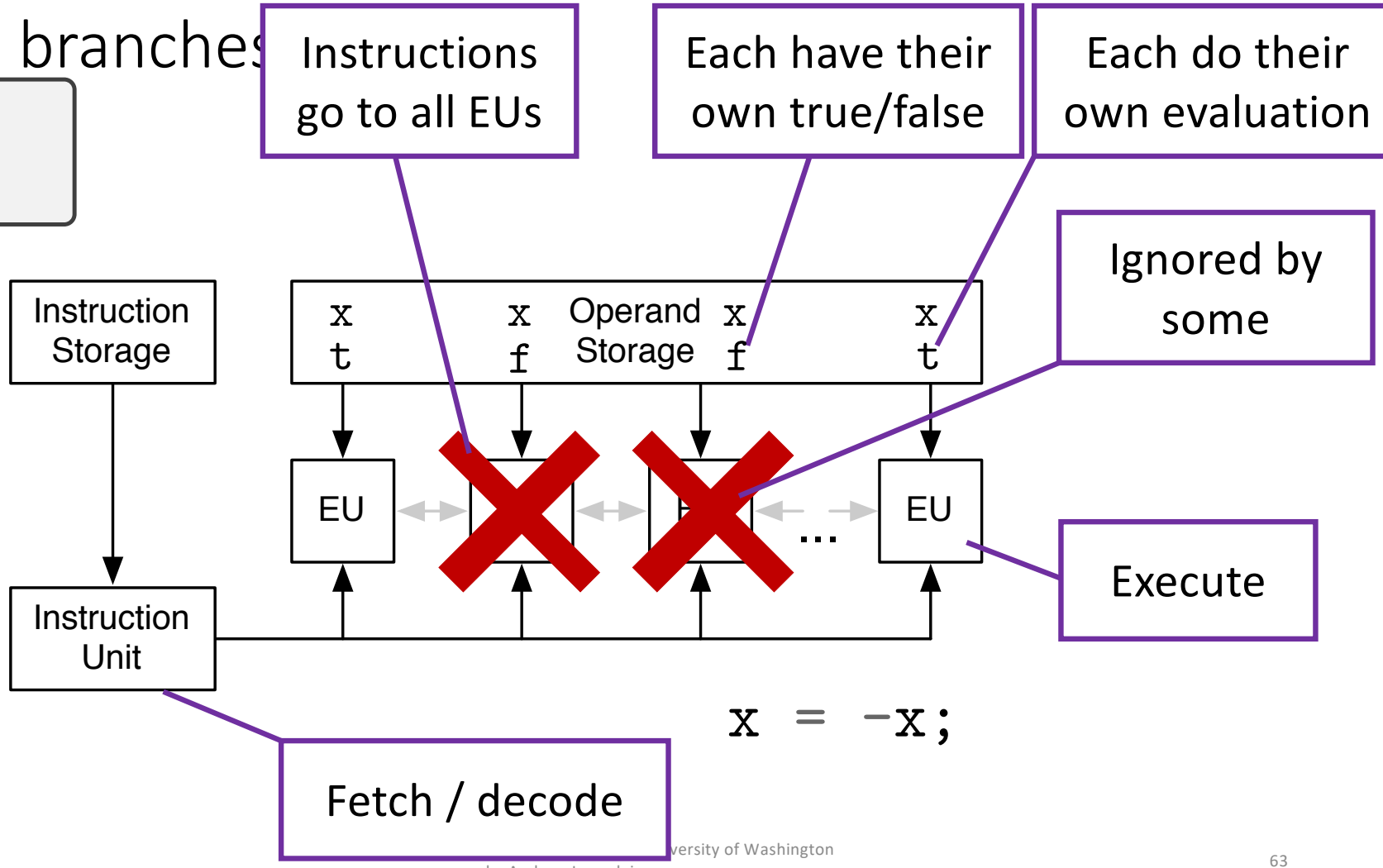


Execute

Fetch / decode

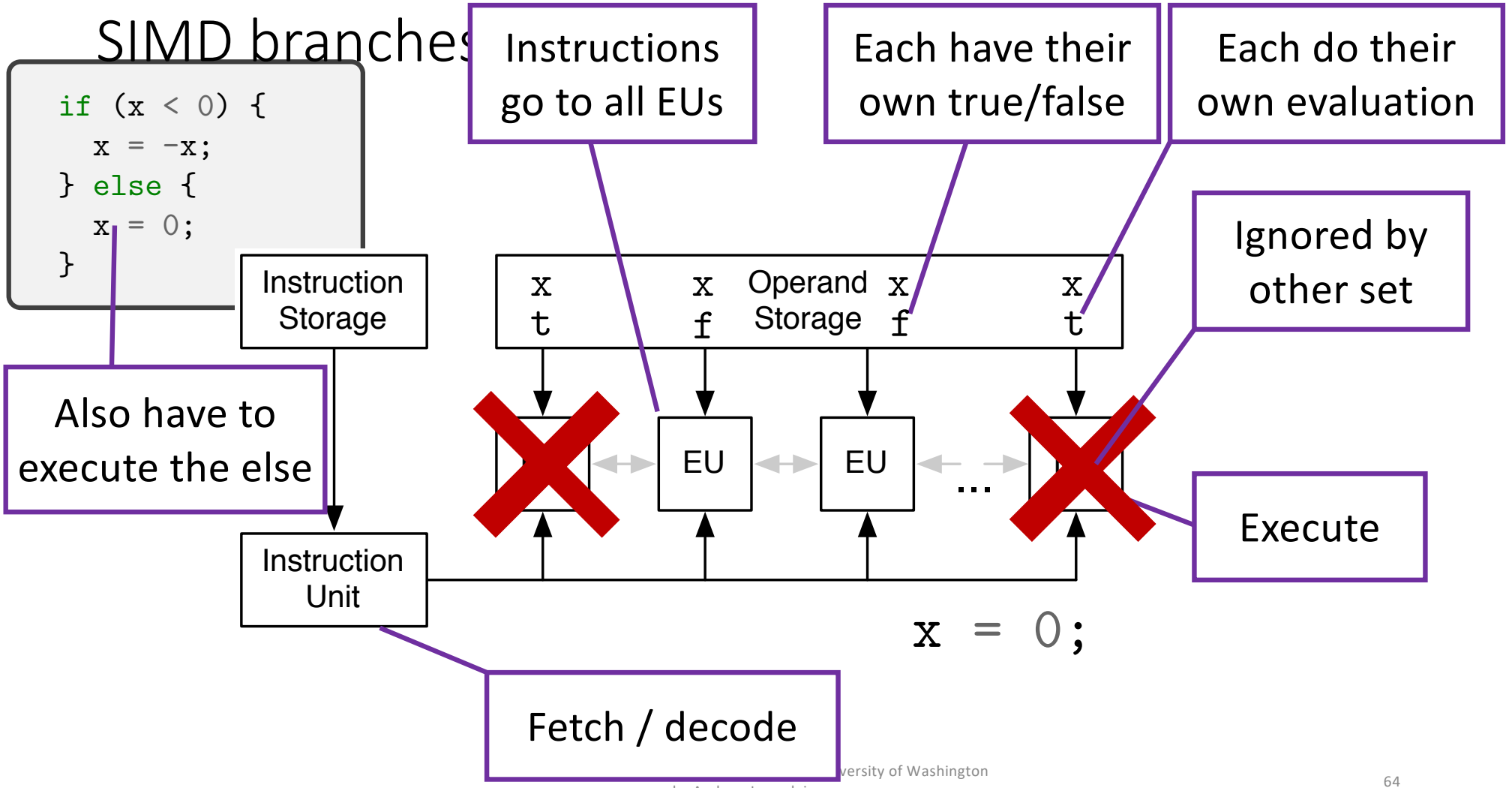
SIMD branches

```
if (x < 0) {
  x = -x;
}
```



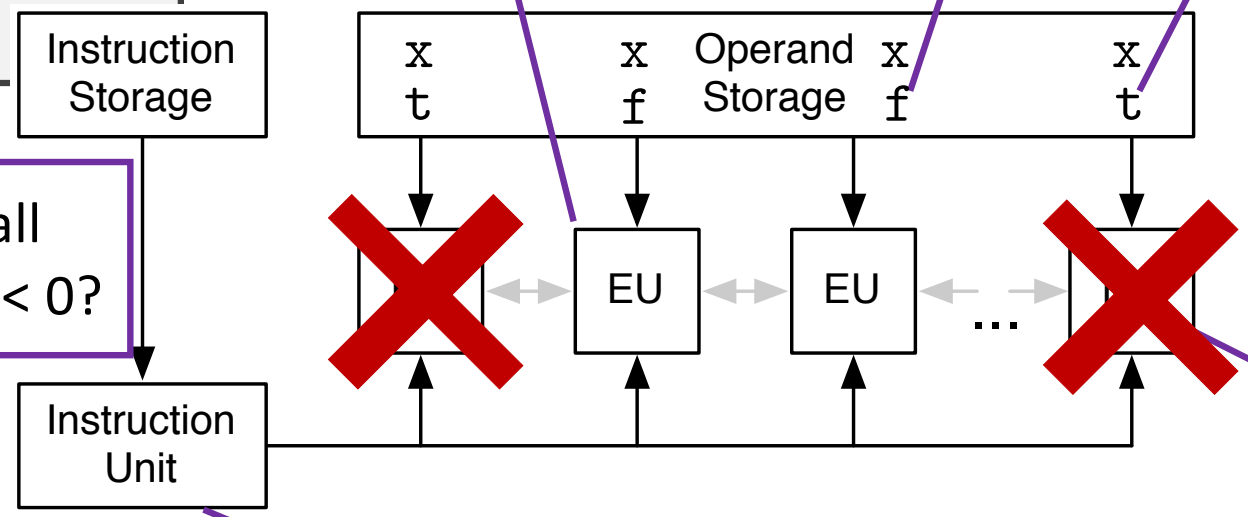
SIMD branches

```
if (x < 0) {
  x = -x;
} else {
  x = 0;
}
```



SIMD branches

```
if (x < 0) {
  x = -x;
} else {
  x = 0;
}
```



Instructions go to all EUs

Each have their own true/false

Each do their own evaluation

What if all values of x < 0?

Executed **both** branches

Still execute **both** branches

Fetch / decode

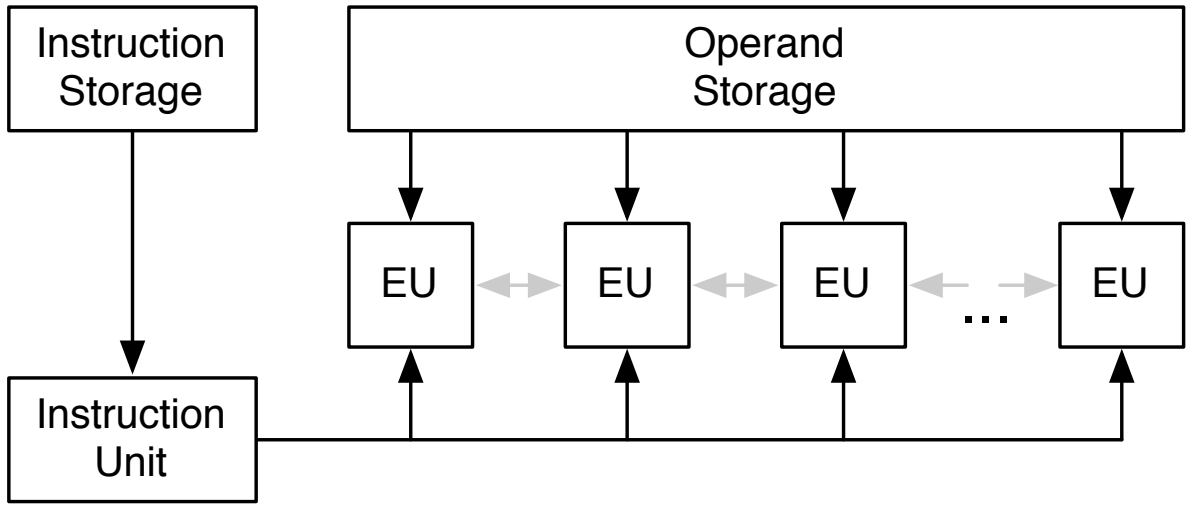
"Divergence"

Bad

SIMD branches

```
if (x < 0) {
  x = -x;
} else {
  x = 0;
}
```

What if we have to get x from memory?



CUDA

```
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20; // 1M elements

  float *x = new float[N];  float *y = new float[N];

  for (int i = 0; i < N; i++)
    y[i] = 2* x[i] = 1.0f;

  add(N, x, y);

  // ...

  delete [] x;  delete [] y;

  return 0;
}
```

Pointers! Bad!

No RAII! Bad!

CUDA

```
__global__
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
    y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1 << 20;
  float *x = nullptr, *y = nullptr;

  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  for (int i = 0; i < N; i++)
    y[i] = 2.0 * x[i] = 1.0f;

  add<<<1, 1>>>(N, x, y);

  cudaDeviceSynchronize();

  cudaFree(x);  cudaFree(y);

  return 0;
}
```

This is a CUDA kernel

Pointers! Bad!

Malloc! Bad!

Execute on device

Compile with nvcc

Execution configuration

No RAII! Bad!

Before

```
__global__
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
    y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1 << 20;
  float *x = nullptr, *y = nullptr;

  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  for (int i = 0; i < N; i++)
    y[i] = 2.0 * x[i] = 1.0f;

  add<<<1, 1>>>(N, x, y);

  cudaDeviceSynchronize();

  cudaFree(x);  cudaFree(y);

  return 0;
}
```

After

```
__global__
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
    y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1 << 20;
  float *x = nullptr, *y = nullptr;

  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  for (int i = 0; i < N; i++)
    y[i] = 2.0 * x[i] = 1.0f;

  add<<<1, 256>>>(N, x, y);

  cudaDeviceSynchronize();

  cudaFree(x);  cudaFree(y);

  return 0;
}
```

Every thread sees same instructions

Blocks

Threads

Do we want 256 threads doing this?

Recall partitioned two norm

Need begin, end (and stride)

Partitioned

Threads are identical except for one thing

How many threads will execute kernel?

How many blocks will execute kernel?

```
__global__
void add(int n, float *x, float *y)
{
  int index = threadIdx.x;
  int stride = blockDim.x;
  for (int i = index; i < n; i += stride)
      y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1 << 20;
  float *x = nullptr, *y = nullptr;

  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  for (int i = 0; i < N; i++)
    y[i] = 2.0 * x[i] = 1.0f;

  add<<<1, 256>>>(N, x, y);

  cudaDeviceSynchronize();

  cudaFree(x);  cudaFree(y);

  return 0;
}
```

Index of current thread in block

Size of block

Partitioned, strided, iteration

Blocks

Needs to be modified

Multiple blocks

```
__global__
void add(int n, float *x, float *y) {
  int index = threadIdx.x, stride = blockDim.x;
  for (int i = index; i < n; i += stride)
      y[i] = x[i] + y[i];
}

int main() {
  int N = 1 << 20;
  float *x = nullptr, *y = nullptr;

  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  for (int i = 0; i < N; i++)
    y[i] = 2.0 * x[i] = 1.0f;

  int blockSize = 256;
  int numBlocks = (N + blockSize - 1) / blockSize;
  add<<<numBlocks, blockSize>>>(N, x, y);

  cudaDeviceSynchronize();

  cudaFree(x);  cudaFree(y);

  return 0;
}
```

Blocks

Index taking into account blocks

Stride taking into account blocks

Multiple blocks

```
__global__
void add(int n, float *x, float *y){
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[I];
}

int main() {
  int N = 1 << 20;
  float *x = nullptr, *y = nullptr;

  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  for (int i = 0; i < N; i++)
    y[i] = 2.0 * x[i] = 1.0f;

  int blockSize = 256;
  int numBlocks = (N + blockSize - 1) / blockSize;
  add<<<numBlocks, blockSize>>>(N, x, y);

  cudaDeviceSynchronize();
  cudaFree(x);  cudaFree(y);

  return 0;
}
```

Kernel Launch

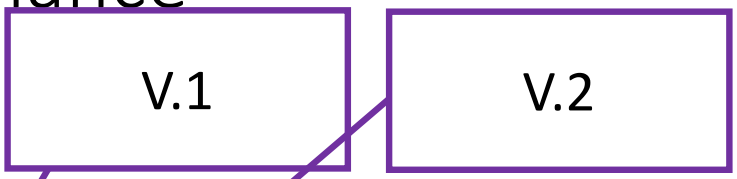
Synchronous or asynchronous?

Hidden memcpy

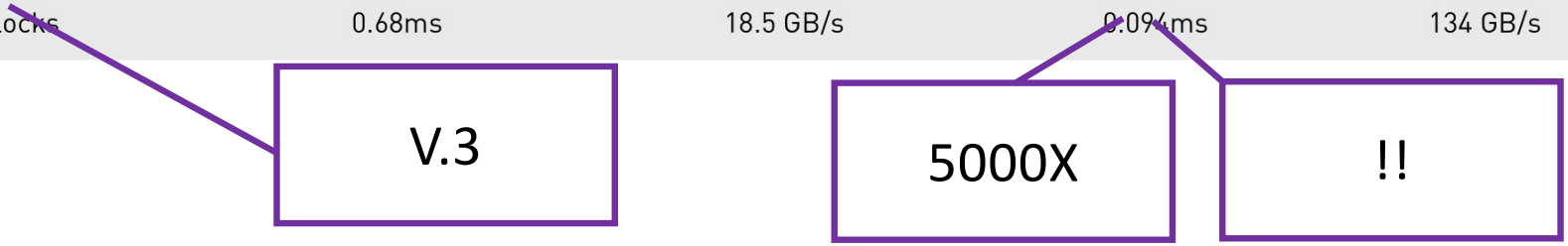
Synchronous or asynchronous?

```
__global__
void add(int n, float *x, float *y){
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[I];
}

int main() {
  int N = 1 << 20;
  float *x = nullptr, *y = nullptr;

  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  for (int i = 0; i < N; i++)
    y[i] = 2.0 * x[i] = 1.0f;

  int blockSize = 256;
  int numBlocks = (N + blockSize - 1) / blockSize;
  add<<<numBlocks, blockSize>>>(N, x, y);

  cudaDeviceSynchronize();
  cudaFree(x);  cudaFree(y);

  return 0;
}
```

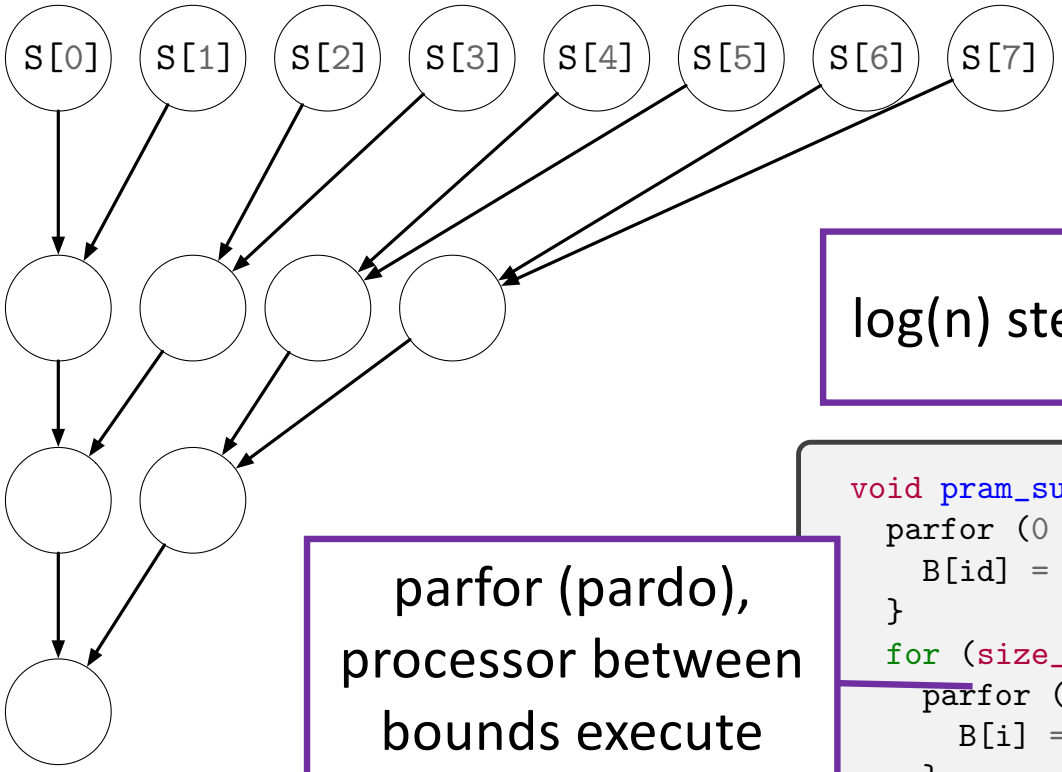
Performance



Version	Laptop (GeForce GT 750M)		Server (Tesla K80)	
	Time	Bandwidth	Time	Bandwidth
1 CUDA Thread	411ms	30.6 MB/s	463ms	27.2 MB/s
1 CUDA Block	3.2ms	3.9 GB/s	2.7ms	4.7 GB/s
Many CUDA Blocks	0.68ms	18.5 GB/s	0.094ms	134 GB/s



PRAM Sum



Each statement in parfor is one parallel step

log(n) steps!

parfor (pardo), processor between bounds execute body

```
void pram_sum () {
  parfor (0 <= id < N) {
    B[id] = A[id];
  }
  for (size_t h = 0; h < log2(N); ++h) {
    parfor (0 <= id < (2<<h)) {
      B[i] = B[2i] + B[2*i+1];
    }
  }
}
```

PRAM -> CUDA

```
void pram_sum () {
  parfor (0 <= id < N) {
    B[id] = A[id];
  }
  for (size_t h = 0; h < log2(N); ++h) {
    parfor (0 <= id < (2<<h)) {
      B[id] = B[2*id] + B[2*id+1];
    }
  }
}
```

Don't need parfor (why?)

Don't need parfor (why?)

```
void pram_sum () {
  size_t tid = threadIdx.x;
  size_t id  = blockIdx.x*blockDim.x + threadIdx.x;
  B[id] = A[id];

  for (size_t h = 0; h < log2(N); ++h) {
    B[id] = B[2*id] + B[2*id+1];
  }
}
```

PRAM -> CUDA

```
void pram_sum () {
  size_t tid = threadIdx.x;
  size_t id  = blockIdx.x*blockDim.x + threadIdx.x;
  B[id] = A[id];

  for (size_t h = 0; h < log2(N); ++h) {
    B[id] = B[2*id] + B[2*id+1];
  }
}
```

Real interface

```
__global__ void reduce0(int *g_idata, int *g_odata) {

  unsigned int tid = threadIdx.x;
  unsigned int id = blockIdx.x*blockDim.x + threadIdx.x;

  B[id] = A[id];

  for (size_t h = 0; h < log2(N); ++h) {
    B[id] = B[2*id] + B[2*id+1];
  }
}
```

PRAM -> CUDA

```
__global__ void reduce0(int *g_idata, int *g_odata) {

  unsigned int tid = threadIdx.x;
  unsigned int id = blockIdx.x*blockDim.x + threadIdx.x;

  B[id] = A[id];

  for (size_t h = 0; h < lo
    B[id] = B[2*id] + B[2*i
  }
}
```

Use shared memory rather than global

```
__global__ void reduce0(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];

  unsigned int tid = threadIdx.x;
  unsigned int id = blockIdx.x*blockDim.x + threadIdx.x;

  sdata[tid] = g_idata[id];

  for (size_t h = 0; h < log2(blockDim.x); ++h) {
    sdata[tid] = sdata[2*tid] + sdata[2*tid+1];
  }
}
```

Danger!

PRAM -> CUDA

```
__global__ void reduce0(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];

  unsigned int tid = threadIdx.x;
  unsigned int id = blockIdx.x*blockDim.x + threadIdx.x;

  sdata[tid] = g_idata[id];

  for (size_t h = 0; h < log2(block
    sdata[tid] = sdata[2*tid] + sda
  }
}
```

```
__global__ void reduce0(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];

  unsigned int tid = threadIdx.x;
  unsigned int id = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[tid] = g_idata[id];
  __syncthreads();

  for (size_t h = 0; h < log2(blockDim.x); ++h) {
    sdata[tid] = sdata[2*tid] + sdata[2*tid+1];
  }
  if (tid == 0)
    g_odata[blockIdx.x] = sdata[0];
}
```

Copy answer back out

I thought threads were synchronized?

Danger!

Reduction #1

```
__global__ void reduce0(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];

  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[tid] = g_idata[i];
  __syncthreads();

  for (size_t s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }

  if (tid == 0)
    g_odata[blockIdx.x] = sdata[0];
}
```

Each thread loads one element from global to shared mem

Reduction is done in shared memory

sync

sync

A different reduction tree

https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Reduction #1

```
__global__ void reduce0(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];

  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[tid] = g_idata[i];
  __syncthreads();

  for (size_t s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }

  if (tid == 0)
    g_odata[blockIdx.x] = sdata[0];
}
```

Several things "wrong"

Divergence, operator %

Memory bank conflicts

https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Reductic

```
__global__ void reduce0(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];

  size_t tid = threadIdx.x;
  size_t i = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[tid] = g_idata[i];
  __syncthreads();

  for (size_t s=1; s < blockDim.x; s *= 2) {
    size_t index = 2 * s * tid;
    if (index < blockDim.x) {
      sdata[index] += sdata[index + s];
     }
     __syncthreads();
  }


  if (tid == 0)
    g_odata[blockIdx.x] = sdata[0];
}
```

Non-divergent

Strided index

Causes memory bank conflicts

Reduction

```
__global__ void reduce0(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];

  size_t tid = threadIdx.x;
  size_t i = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[tid] = g_idata[i];
  __syncthreads();

  for (size_t s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }

  if (tid == 0)
    g_odata[blockIdx.x] = sdata[0];
}
```

Idle threads

Iterate decreasing

tid based indexing (contiguous)

Final Reduction

```
__global__ void reduce0(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];

  size_t tid = threadIdx.x;
  size_t i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
  sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
  __syncthreads();

  for (size_t s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }

  if (tid == 0)
    g_odata[blockIdx.x] = sdata[0];
}
```

Do first iteration on load

Results

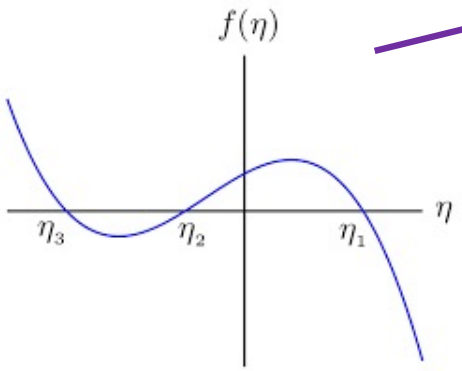
	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x

Gflops?

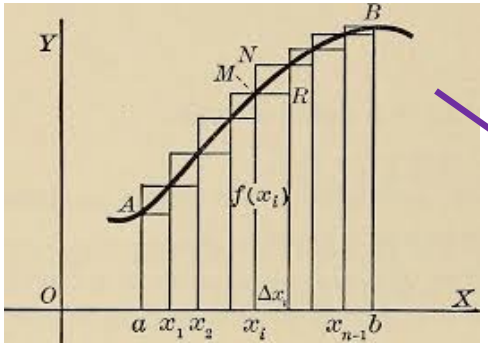
Fundamental Theorem

$$N = \prod_{i=0}^{m} x_i$$

Arithmetic: Every number is a product of primes



Algebra: Every polynomial has a root



$$\frac{\mathrm{d}}{\mathrm{dt}} \int_0^t f(t)dt = \int_0^t \left[\frac{\mathrm{d}}{\mathrm{dt}} f(t)\right] = f(t)$$

Calculus: The integral of the derivative is the derivative of the integral

Software engineering: We can solve any problem by introducing an extra level of indirection

David Wheeler via Butler Lampson via Andrew Koenig

Linear Systems

$$x = \sum_{i=0}^{\dim X} \alpha_i y_i$$

Every linear space has a basis

Any element in the space can be expressed as weighted sums of members of the basis

$$e_i = (0, 0, \ldots 1 \ldots 0)$$

Nice orthonormal basis

$$x = (x_0, x_1 \ldots x_{N-1})$$

$$\hat{x} = (\alpha_0, \alpha_1, \ldots \alpha_{N-1})$$

The same element has multiple representations

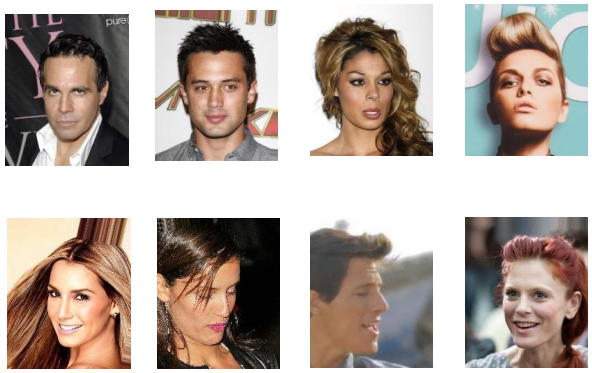
Name This Famous Person

Name This Famous Person

Principal Components Analysis

- We are given a training set X of faces
- We want to find an orthonormal basis that can form an alternate representation of faces with as few dimensions as possible
  - Axes are the “principal components”
  - First axis captures as much of the data set as possible
  - Next axis captures as much of the data that isn’t captured by first
  - And so on
- We can represent any face with linear combination of the principal components

Principal Component Analysis



Training set

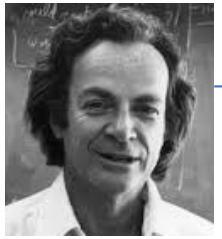
Mean

[C]

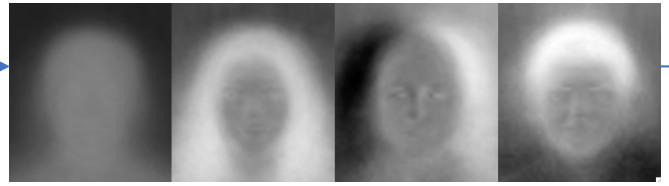
Covariance

Principal components

Principal Component Analysis



Project face onto principal components

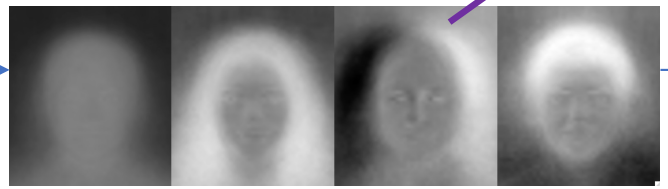


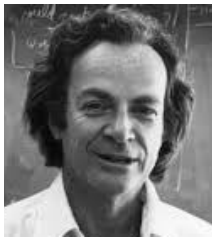
Representation in feature space

$$\{\phi_0, \phi_1, \ldots, \phi_{N-1}\}$$

How do we compute these?

$$\{\phi_0, \phi_1, \ldots, \phi_{N-1}\}$$





Linear combination of principal components

Recreate original face

Computing Principal Components

Let $\{f_0, f_1, \ldots f_M\}$ be as set of faces. Each face $f_i \in \mathbb{R}^N$ where N is the total number of pixels in a face image and the elements of each f_i all have the same correspond to the pixels in face image i (without loss of generality, take lexicographical ordering). Let $\mathbf{F} \in \mathbb{R}^{M \times N}$ be a matrix in which every column i consists of f_i.

Let $\Phi \in \mathbb{R}^{N \times N}$ be an orthonormal matrix where each column is a feature vector ϕ_i. For a given face f_i we let $\tilde{f}_i = \sum_{j=0}^{K-1} \alpha_{i,j}\phi_j$ and define Φ such that for each K the difference between f_i and $\tilde{f}_i$ is minimized for $i = 0, 1, \ldots, N-1$.

Let's start with the case of $K = 0$. Then each $\tilde{f}_i = \alpha_i \phi_0$. The best approximation of f_i will be the projection of f_i onto ϕ_0, i.e., $\alpha_i = \langle f_i, \phi_0 \rangle$.

Computing Principal Components

The sum of squares difference between all of the f_i and their projection onto ϕ_0 is

$$\sum_{i=0}^{N-1} \| f_i - \langle f_i, \phi_0 \rangle \phi_0 \|_2^2$$

The best choice for ϕ_0 is thus the one that minimizes this expression, i.e.,

$$\phi_0 = \operatorname{argmin} \sum_{i=0}^{N-1} \| f_i - \langle f_i, \phi_0 \rangle \phi_0 \|_2^2 = \operatorname{argmin} \sum_{i=0}^{N-1} \langle f_i - \langle f_i, \phi_0 \rangle \phi_0, f_i - \langle f_i, \phi_0 \rangle \phi_0 \rangle$$

$$= \operatorname{argmin} \sum_{i=0}^{N-1} (\langle f_i, f_i \rangle - \langle f_i, \langle f_i, \phi_0 \rangle \phi_0 \rangle)$$

Computing Principal Components

$$\phi_0 = \text{argmax} \sum_{i=0}^{N-1} \langle f_i, \langle f_i, \phi_0 \rangle \phi_0 \rangle = \text{argmax} \sum_{i=0}^{N-1} \langle \langle \phi_0, f_i \rangle f_i, \phi_0 \rangle$$

$$= \text{argmax} \sum_{i=0}^{N-1} \phi_0^T f_i f_i^T \phi_0 = \text{argmax} \left[\phi_0^T \left(\sum_{i=0}^{N-1} f_i f_i^T \right) \phi_0 \right]$$

$$= \text{argmax} \left(\langle \phi_0, C\phi_0 \rangle \right)$$

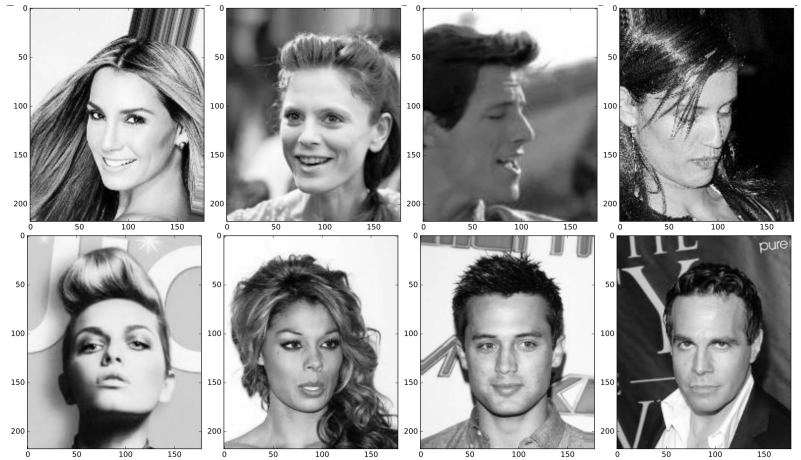
Covariance Matrix

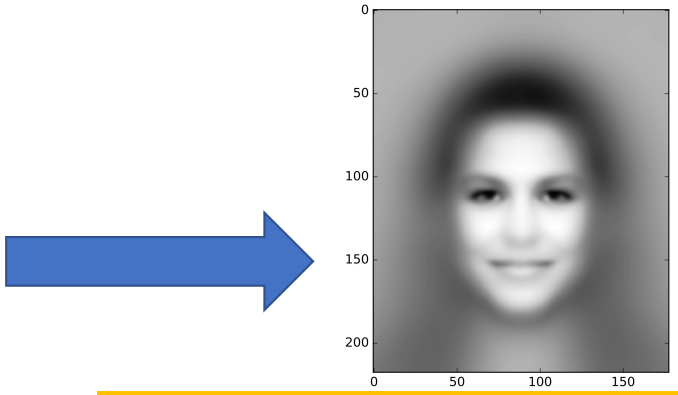
Rayleigh Quotient

Summary (PCA)

- Compute covariance matrix
  - Compute mean of all vectors (faces)
  - Subtract mean from all vectors (center the data)
  - Compute covariance matrix (sum of outer products of each face with itself)
- Compute SVD of covariance matrix

Mean Celebrity Face Computation



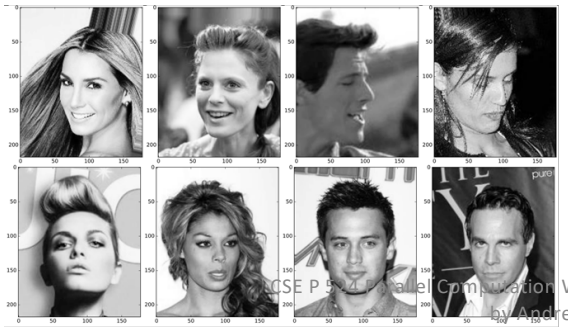


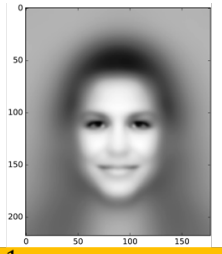
$$\mu_j = \frac{1}{m}\sum_{i=0}^{m-1} v_j^{(i)} \text{ for all } j \in \{0, \ldots, n-1\}$$

- $v^{(i)} \in \mathbb{R}^n$
  - Image of the CelebA database of size n = 45×55 = 2475
  - Total of m = 202,599 images
- $v_j^{(i)} \in \mathbb{R}$
  - **Pixel *j*** of **image *i***
- $D = \left(v^{(0)}, v^{(1)}, \ldots, v^{(m-1)}\right) \in \mathbb{R}^{m \times n}$
  - Data matrix with total volume: (45×55) × sizeof(float) × 202,599 ≈ 1.8 GB

Mean Celebrity Face Computation

```
// Header of the mean computation program

#include "../include/hpc_helpers.hpp" // timers, error macros
#include "../include/bitmap_IO.hpp"   // write images
#include "../include/binary_IO.hpp"   // load data

template <
    typename index_t,                 // data type for indices
    typename value_t> __global__      // data type for values
void compute_mean_kernel(
    value_t * Data,                   // device pointer to data
    value_t * Mean,                   // device pointer to mean
    index_t num_entries,              // number of images (m)
    index_t num_features);            // number of pixels (n)
```

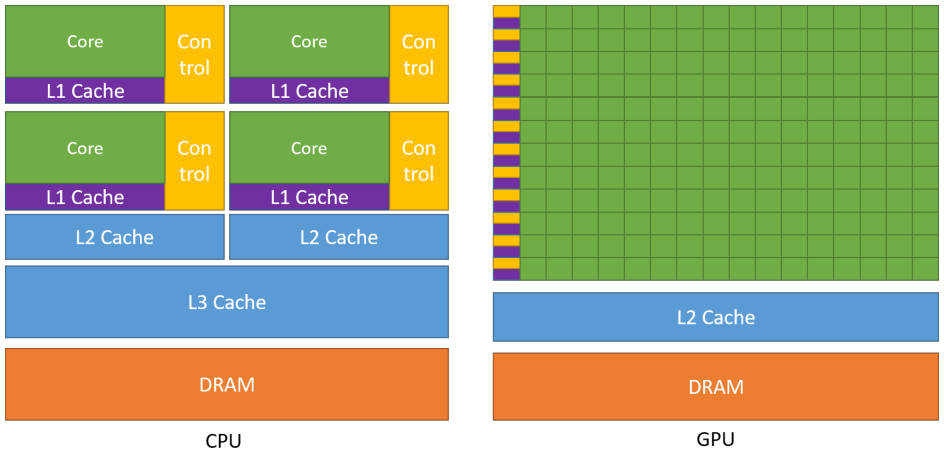


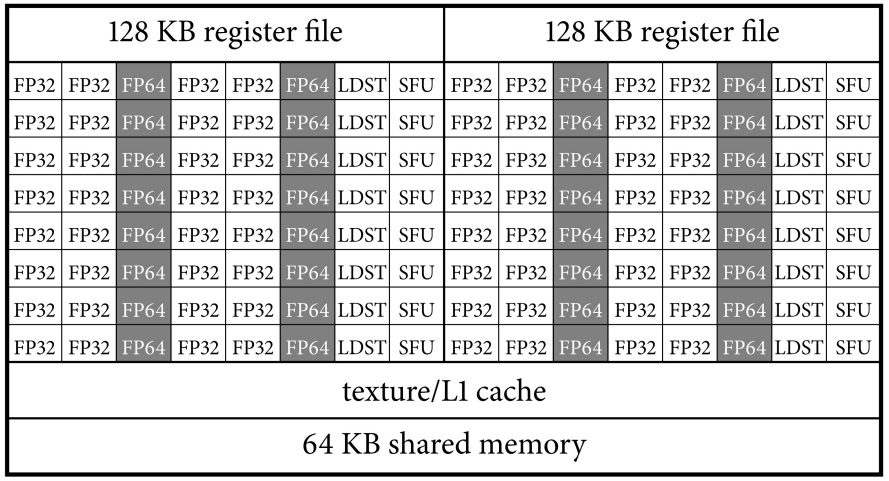


$$\mu_j = \frac{1}{m}\sum_{i=0}^{m-1} v_j^{(i)} \quad \text{for all } j \in \{0, \ldots, n-1\}$$

Thread and Memory Hierarchy

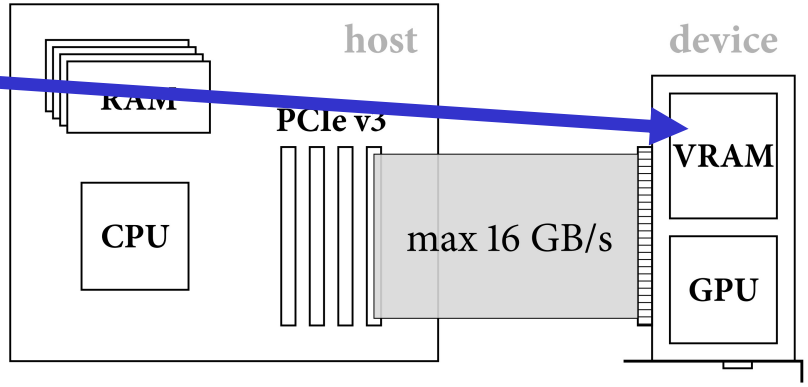
- Each thread has its own **local memory / registers**
- **Shared memory** is separate per thread block
- **Global memory** accessible by all threads





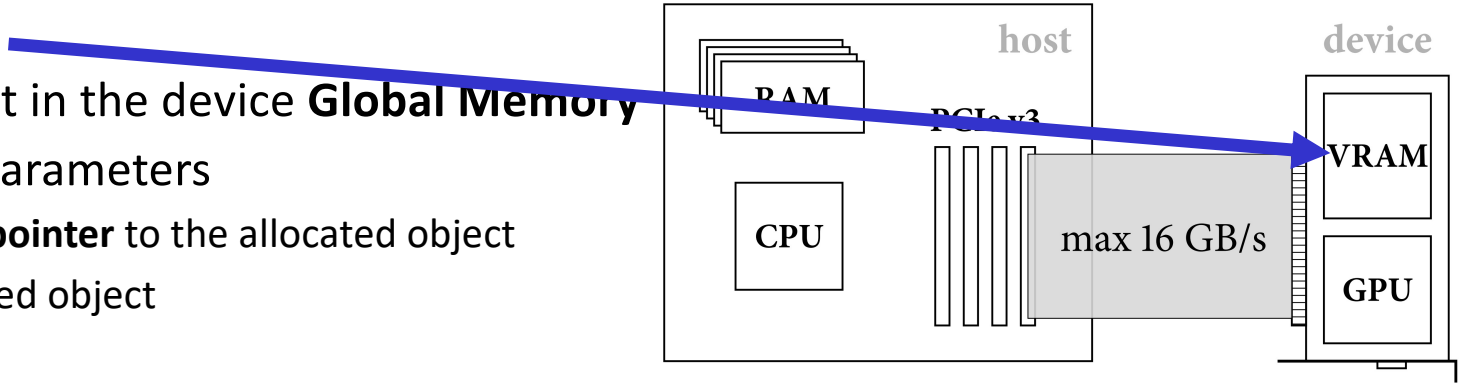
CUDA Memory Model Overview

- Global memory
  - Main means of communicating R/W Data between **host** and **device**
  - Contents visible to all threads
  - Long latency access
- We will focus on global memory for now
  - Constant and texture memory for advanced usage



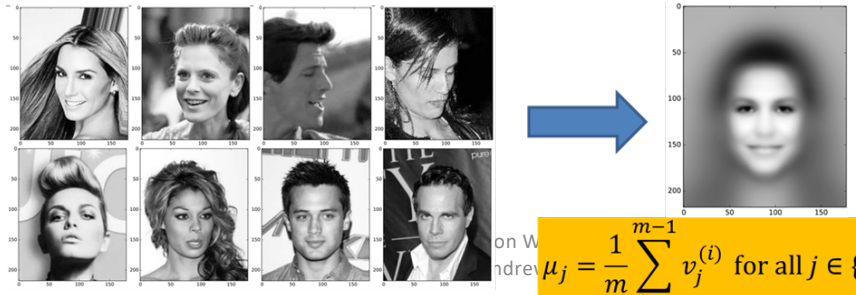
CUDA Device Memory Allocation

- **cudaMalloc()**
  - Allocates object in the device **Global Memory**
  - Requires two parameters
    - **Address of a pointer** to the allocated object
    - **Size** of allocated object
- **cudaFree()**
  - Frees object from device Global Memory



Mean Celebrity Face Computation

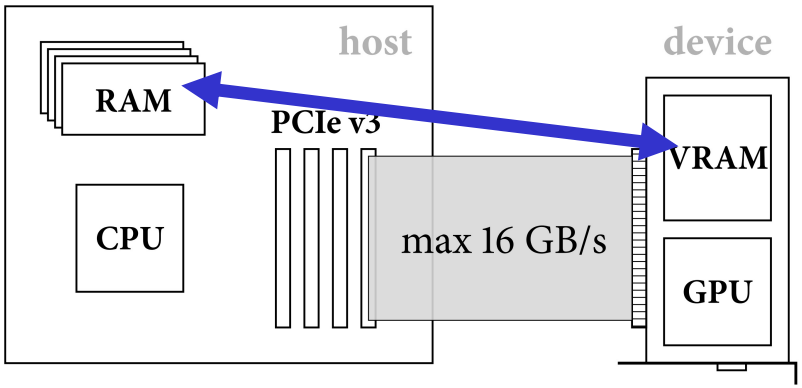
```
// CPU Main function: memory allocation and loading data from disk
int main(int argc, char * argv[]) {
    cudaSetDevice(0);
    // 202599 grayscale images each of shape 55 x 45
    constexpr uint64_t imgs = 202599, rows = 55, cols = 45;

    // pointer for data matrix and mean vector on CPU
    float *data = nullptr, *mean = nullptr;
    cudaMallocHost(&data, sizeof(float)*imgs*rows*cols);
    cudaMallocHost(&mean, sizeof(float)*rows*cols);

    // allocate storage on GPU
    float *Data = nullptr, *Mean = nullptr;
    cudaMalloc(&Data, sizeof(float)*imgs*rows*cols);
    cudaMalloc(&Mean, sizeof(float)*rows*cols);

    // load data matrix from disk
    std::string file_name =  "./data/celebA_gray_lowres.202599_55_45_32.bin";
    load_binary(data, imgs*rows*cols, file_name);
```



$$\mu_j = \frac{1}{m}\sum_{i=0}^{m-1} v_j^{(i)} \text{ for all } j \in \{0, \ldots, n-1\}$$

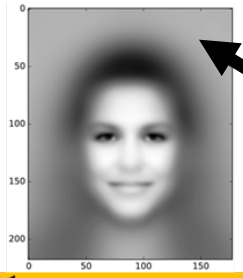
CUDA Host-Device Data Transfer

- **cudaMemcpy()**
  - memory data transfer
  - requires 4 parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - **Type of transfer**
      - Host to Host,
      - Host to Device
      - Device to Host
      - Device to Device
  - ***blocks*** host execution until all preceding tasks have completed




```
#define H2D (cudaMemcpyHostToDevice)
#define D2H (cudaMemcpyDeviceToHost)
```

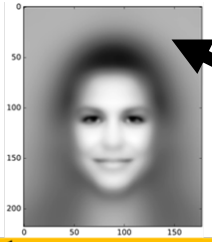
```
    // Memory transfers and kernel invocation
    // Copy data to device and reset Mean
    cudaMemcpy(Data, data, sizeof(float)*imgs*rows*cols, H2D);
    cudaMemset(Mean, 0, sizeof(float)*rows*cols);

    // Compute mean
    compute_mean_kernel <<<SDIV(rows*cols,32), 32 >>>(Data, Mean, imgs, rows*cols);

    // Transfer mean back to host
    cudaMemcpy(mean, Mean, sizeof(float)*rows*cols, D2H);

    // Writing the result and freeing memory
    dump_bitmap(mean, rows, cols, "./imgs/celebA_mean.bmp");

    cudaFreeHost(data); cudaFreeHost(mean); // get rid of the memory
    cudaFree(Data); cudaFree(Mean);
}
```



1 thread per pixel in mean image. Use 32 threads per thread block

$$\mu_j = \frac{1}{m}\sum_{i=0}^{m-1} v_j^{(i)} \text{ for all } j \in \{0, \ldots, n-1\}$$

```
// CUDA kernel implementing the mean face computation
template <typename index_t, typename value_t> __global__
void compute_mean_kernel(
    value_t * Data, value_t * Mean, index_t num_entries, index_t num_features) {

    // compute global thread identifier
    const auto thid = blockDim.x*blockIdx.x + threadIdx.x;

    // prevent memory access violations
    if (thid < num_features) {
        // accumulate in a fast register, not in slow global memory
        value_t accum = 0;
        for (index_t entry = 0; entry < num_entries; entry++)
            accum += Data[entry*num_features + thid];

        // write the register once to global memory
        Mean[thid] = accum / num_entries;
    }
}
```



$$\mu_j = \frac{1}{m}\sum_{i=0}^{m-1} v_j^{(i)} \text{ for all } j \in \{0, \ldots, n-1\}$$

Thread with ID `thid` computes **`mean[thid]`**. Alltogether there are **2496** threads working in parallel partitioned into 78 thread blocks each of size 32.

Mean Celebrity Face Computation

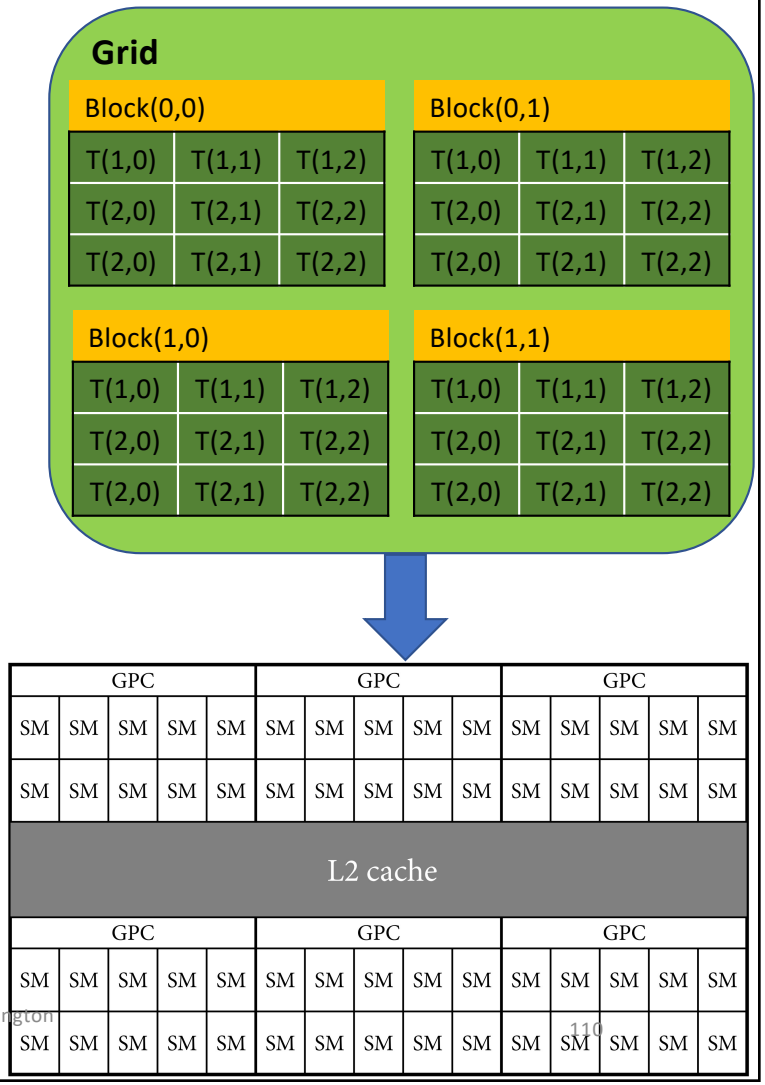
- Code compilation:
  - `nvcc -02 -std=c++11 -arch=sm_61 -o mean_computation`
- Execution times on a Titan X GPU:
  - `TIMING: 170.136 ms (data_H2D)`
  - `TIMING: 8.82074 ms (compute_mean_kernel)`
  - `TIMING: 0.03174 ms (mean_D2H)`

How many Gflops?

- Remarks
  - memory transfer from host to device over the PCIe bus takes 170 ms (≈11 GB/s)
  - CUDA kernel computation accesses each pixel of the dataset exactly once and applies one addition. Thus, 9 ms correspond to an effective global memory bandwidth of approximately 208 GB/s and a compute performance of about 52 GFlop/s
  - Time for transferring the average image (≈9.5 KB) back to host is negligible

Mapping Thread Blocks Onto SMs

- All threads in a block execute the same kernel (**SPMD**)
- Programmer declares block:
  - Block up to *max* number concurrent threads
  - Block shape 1D, 2D, or 3D (in threads)
- Threads have ***thread_id*** numbers within block
- Threads in the same block share data and synchronize while doing their share of the work
- Threads in different blocks cannot efficiently cooperate

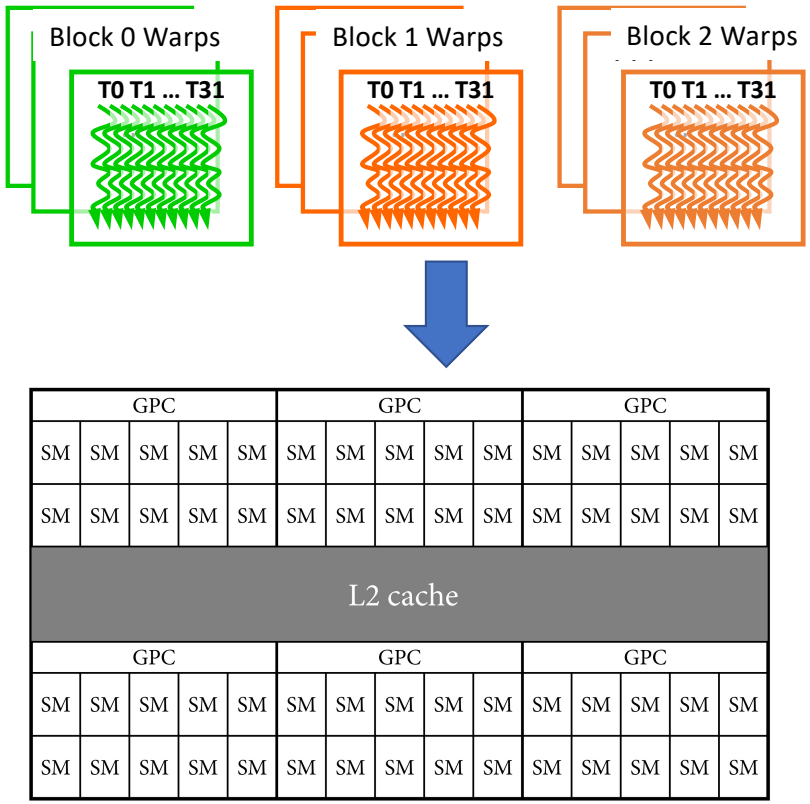


Partitioning of CUDA Thread Blocks into Warps

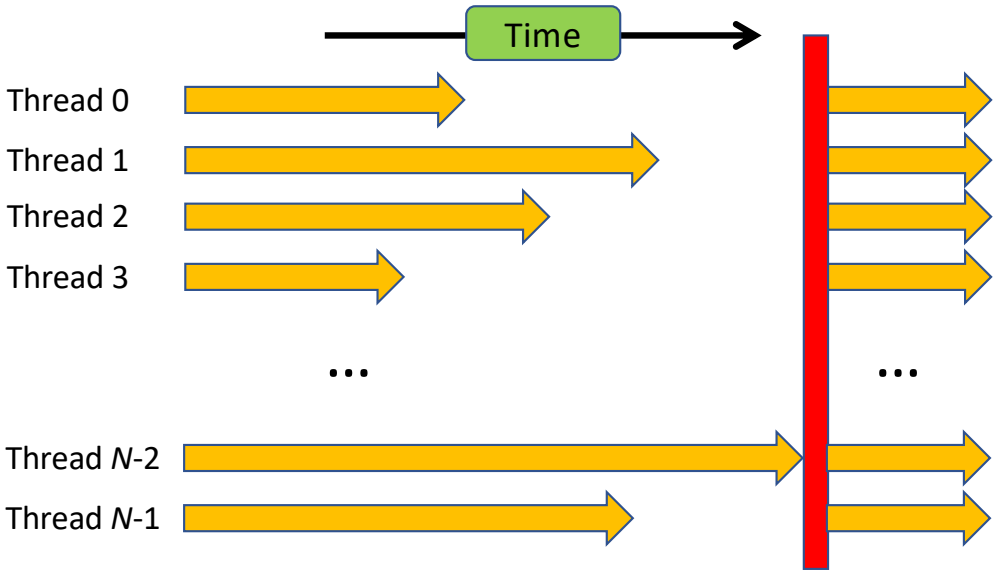
- Thread blocks are partitioned into warps consisting of **32 threads**
  - Thread IDs within a warp are consecutive and increasing
- Partitioning is always the same
  - Thus, you can use this knowledge in control flow
- However, DO NOT rely on any ordering between warps
  - If there are any dependencies between threads, you must ***__syncthreads()*** to get correct results

Warp Scheduling

- SM implements zero-overhead warp scheduling
- At any time, one of the warps is executed by SM
- Warps whose next instruction has its operands ready for consumption are eligible for execution
- Eligible Warps are selected for execution on a prioritized scheduling policy
- **SIMD (Single Instruction Multiple Data)**
  - All threads in a warp execute the same instruction when selected

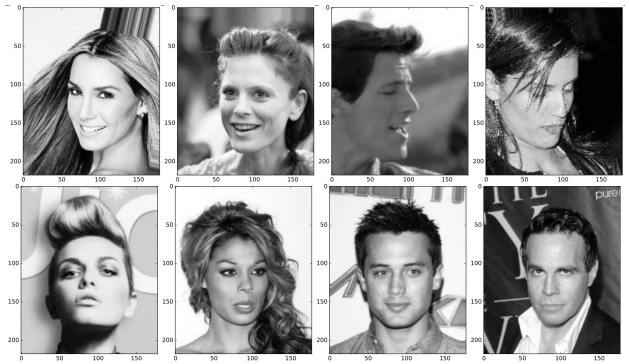


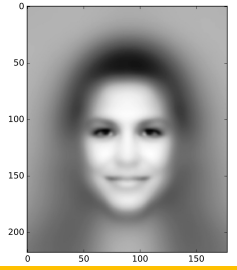
Barrier Synchronization



- ***__syncthreads():*** **All** threads in the **same block** must reach *__syncthreads()* before any can move on
- Used to coordinate tiled algorithms in order to ensure that all elements of a tile are loaded/consumed
- ***__syncthreads()*** allowed in ***If-then-else*** statement but usually makes only sense if conditional evaluates identically across entire thread block, otherwise execution may hang or produce side effects

Computing the Centered Data Matrix





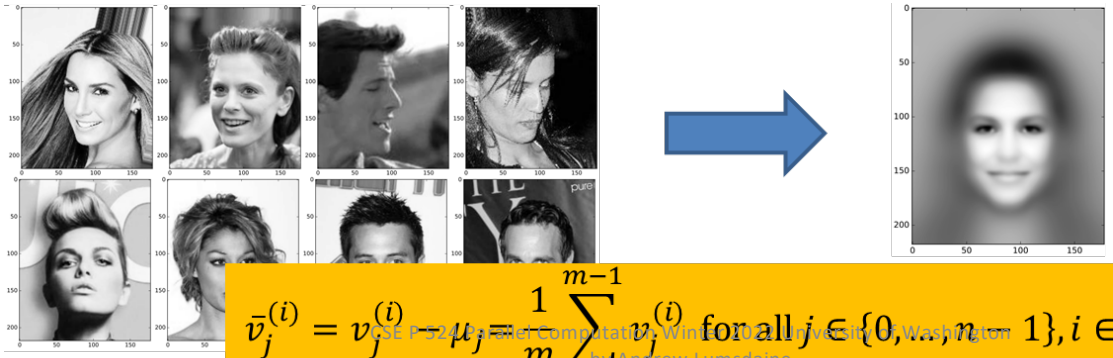
$$\mu_j = \frac{1}{m}\sum_{i=0}^{m-1} v_j^{(i)} \text{ for all } j \in \{0, \dots, n-1\}$$

$$\bar{v}_j^{(i)} = v_j^{(i)} - \mu_j = \frac{1}{m}\sum_{i=0}^{m-1} v_j^{(i)} \text{ for all } j \in \{0, \dots, n-1\}, i \in \{0, \dots, m-1\}$$

- $v^{(i)} \in \mathbb{R}^n$
  - Image of the CelebA database of size *n* = 45×55 = 2475. Total of *m* = 202,599 images
- $v_j^{(i)} \in \mathbb{R}$
  - Pixel *j* of image *i*
- $D = \left(v^{(0)}, v^{(1)}, \dots, v^{(m-1)}\right) \in \mathbb{R}^{m \times n}$
- $\bar{D} = \left(\bar{v}^{(0)}, \bar{v}^{(1)}, \dots, \bar{v}^{(m-1)}\right) \in \mathbb{R}^{m \times n}$
  - *Centered* Data matrix with total volume: (45×55) × sizeof(float) × 202,599 ≈ 1.8 GB

```
//CUDA kernel performing mean correction (parallelizes over pixel index j)
template <typename index_t, typename value_t> __global__
void correction_kernel(value_t * Data, value_t * Mean, index_t num_entries, index_t num_features) {

    const auto thid = blockDim.x*blockIdx.x + threadIdx.x;

    if (thid < num_features) {
        const value_t value = Mean[thid];

        for (index_t entry = 0; entry < num_entries; entry++)
            Data[entry*num_features + thid] -= value;
    }
}
```

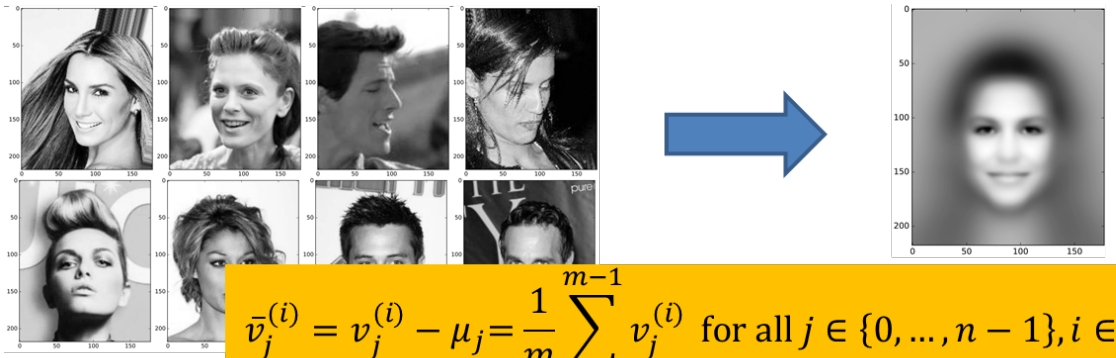
```
correction_kernel<<<SDIV(rows*cols,32),32>>>(Data, Mean, imgs, rows*cols);
```



$$\bar{v}_j^{(i)} = v_j^{(i)} - \mu_j = \frac{1}{m}\sum_{i=0}^{m-1} v_j^{(i)} \quad \text{for all } j \in \{0,\ldots,n-1\}, i \in \{0,\ldots,m-1\}$$

```
//CUDA kernel performing mean correction (parallelizes over image index i)
template < typename index_t, typename value_t> __global__
void correction_kernel_ortho(value_t * Data, value_t * Mean, index_t num_entries, index_t num_features) {

    const auto thid = blockDim.x*blockIdx.x + threadIdx.x;

    if (thid < num_entries) {
        for (index_t feat = 0; feat < num_features; feat++)
            Data[thid*num_features + feat] -= Mean[feat];
    }
}
```

```
correction_kernel_ortho<<<SDIV(imgs,32),32>>>(Data, Mean, imgs, rows*cols);
```

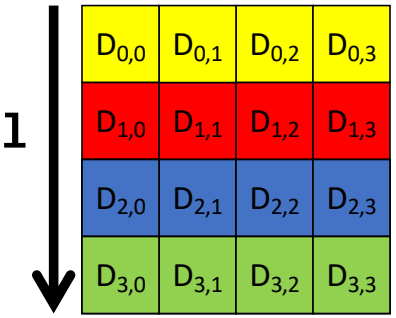


$$\bar{v}_j^{(i)} = v_j^{(i)} - \mu_j = \frac{1}{m}\sum_{i=0}^{m-1} v_j^{(i)} \text{ for all } j \in \{0,\ldots,n-1\}, i \in \{0,\ldots,m-1\}$$

Coalesced Memory Access

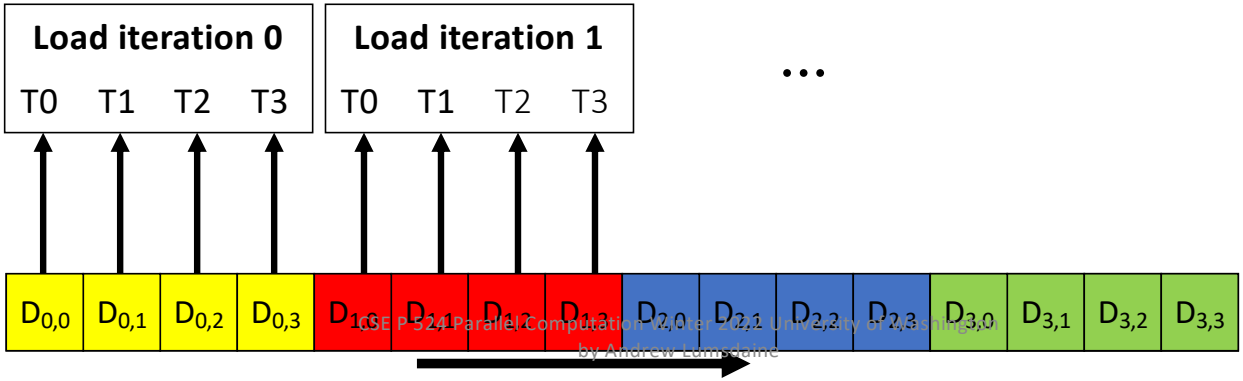
```
for (index_t entry = 0; entry < num_entries; entry++)
            Data[entry*num_features + thid] -= value;
```

Access direction in **`correction_kernel`**



`Data[entry*num_features + thid]`

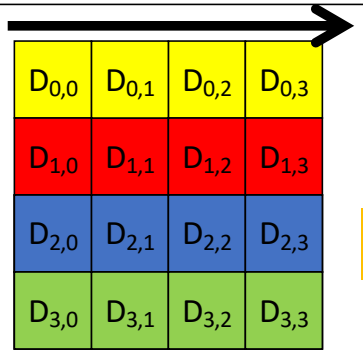
$D_{ij} = v_j^{(i)}$



Non-Coalesced Memory Access

```
for (index_t feat = 0; feat < num_features; feat++)
            Data[thid*num_features + feat] -= Mean[feat];
```

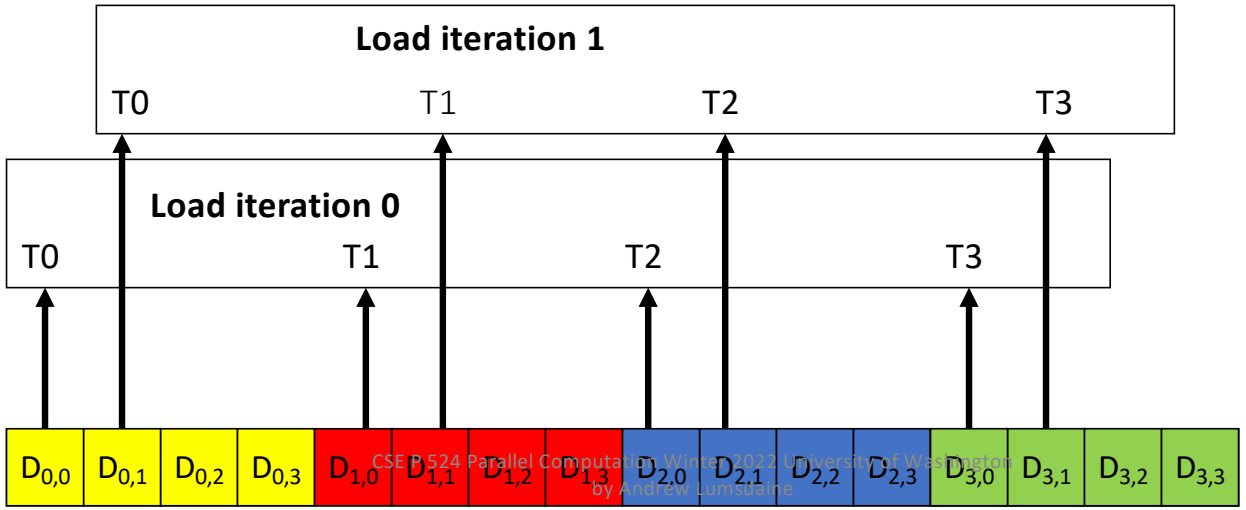
Access direction in **correction_kernel_ortho**



Data[thid*num_features + feat]

$D_{ij} = v_j^{(i)}$

...



The Tale of two Memory Access Patterns

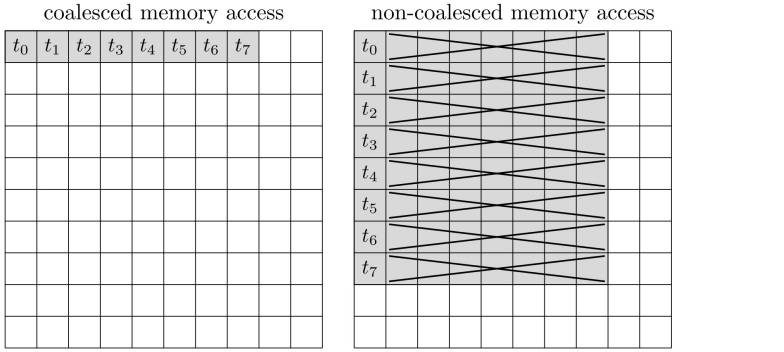


FIGURE 7.7

An example for coalesced and non-coalesced memory access patterns on a 10 × 10 matrix. The left panel shows how eight consecutive threads can simultaneously utilize a whole cache line (gray box). When accessing columns of a matrix as shown in the right panel, each thread issues a load of a complete cache line. Moreover, the remaining seven entries of each cache line are instantly invalidated when manipulating the first entry which enforces a new load during the next iteration. The size of the gray shaded areas corresponds to the expected execution times.

- Measured runtimes on a Titan X:
  - `correction_kernel`: **60 ms**
  - `correction_kernel_ortho`: **500 ms**

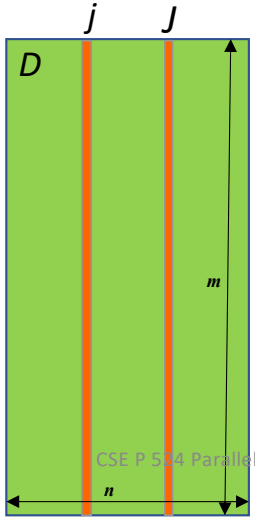
Computation of the Covariance Matrix

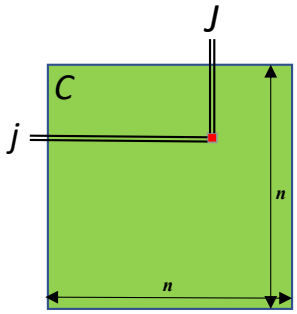
$$C_{jj'} = \frac{1}{m}\sum_{i=0}^{m-1} \bar{v}_j^{(i)} \cdot \bar{v}_{j'}^{(i)} \text{ for all } j, j' \in \{0, \ldots, n-1\}$$

$$C_{jj'} = \frac{1}{m}\left\langle \left(\bar{v}_j^{(0)}, \bar{v}_j^{(1)}, \ldots, \bar{v}_j^{(m-1)}\right) \middle| \left(\bar{v}_{j'}^{(0)}, \bar{v}_{j'}^{(1)}, \ldots, \bar{v}_{j'}^{(m-1)}\right) \right\rangle$$

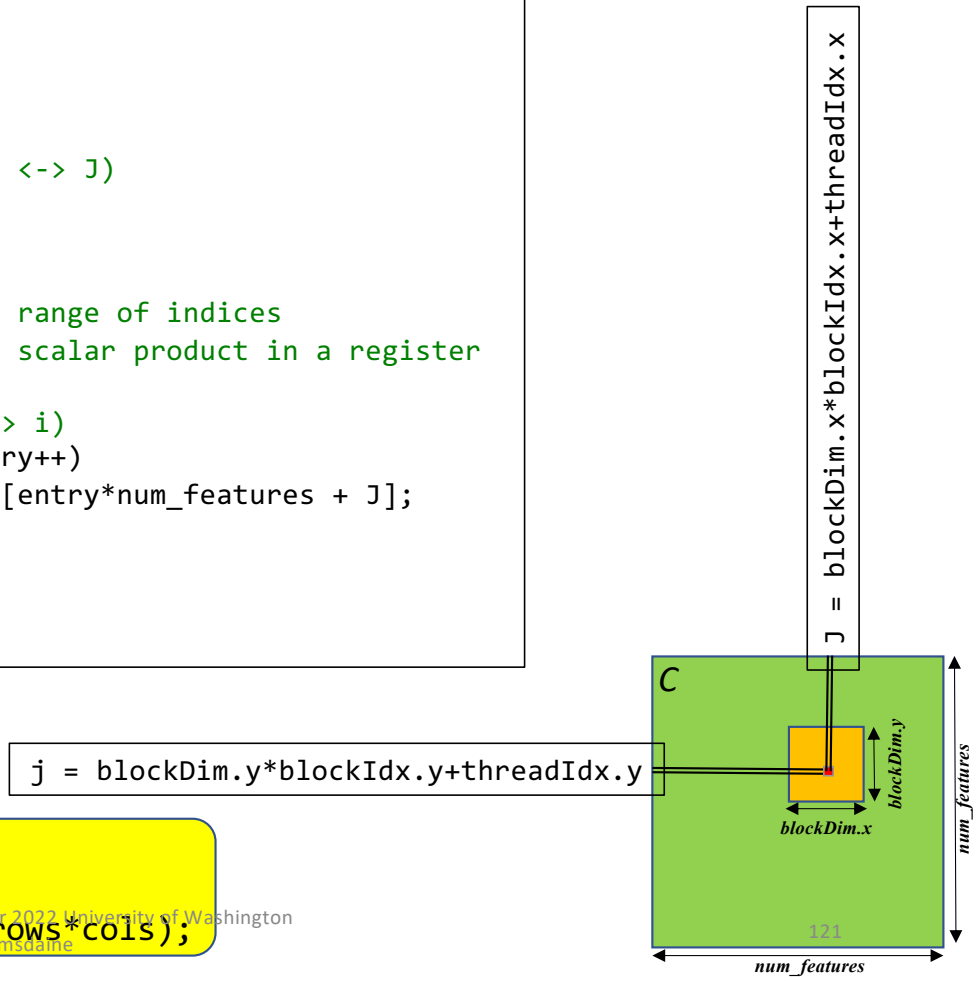
- $\overline{D} = \left(\bar{v}^{(0)}, \bar{v}^{(1)}, \ldots, \bar{v}^{(m-1)}\right) \in \mathbb{R}^{m \times n}$
  - *Centered* Data matrix.
  - Each image of size $n = 45 \times 55 = 2475$. Total of $m = 202{,}599$ images

$D_{ij} = v_j^{(i)}$





```
//CUDA kernel performing naive covariance matrix computation
template <typename index_t, typename value_t>  __global__
void covariance_kernel(
    value_t * Data,          // centered data matrix
    value_t * Cov,           // covariance matrix
    index_t num_entries,     // number of images (m)
    index_t num_features) {  // number of pixels (n)

    // determine row and column indices of Cov (j and j' <-> J)
    const auto J = blockDim.x*blockIdx.x + threadIdx.x;
    const auto j = blockDim.y*blockIdx.y + threadIdx.y;

    if (j < num_features && J < num_features) { // check range of indices
        value_t accum = 0;                      // store scalar product in a register

        // accumulate contribution over images (entry <-> i)
        for (index_t entry = 0; entry < num_entries; entry++)
            accum += Data[entry*num_features + j] * Data[entry*num_features + J];

        Cov[j*num_features + J] = accum / num_entries;
    }
}
```



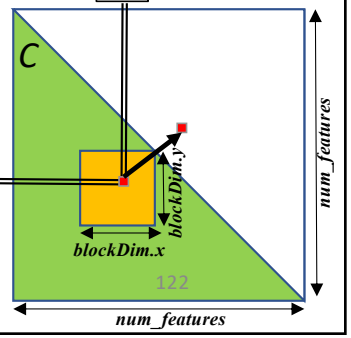

```
dim3 blocks(SDIV(rows*cols,8), SDIV(rows*cols,8));
dim3 threads(8,8);
covariance_kernel<<<blocks,threads>>>(Data,Cov,imgs,rows*cols);
```

```
//CUDA kernel performing symmetric covariance matrix computation
template <typename index_t, typename value_t> __global__
void symmetric_covariance_kernel(
    value_t * Data, value_t * Cov, index_t num_entries, index_t num_features) {

    // indices as before
    const auto J = blockDim.x*blockIdx.x + threadIdx.x;
    const auto j = blockDim.y*blockIdx.y + threadIdx.y;

    // execute only entries below the diagonal since C = C^T
    if (j < num_features && J <= j) {
        value_t accum = 0;

        for (index_t entry = 0; entry < num_entries; entry++)
            accum += Data[entry*num_features + j] * Data[entry*num_features + J];

        // exploit symmetry
        Cov[j*num_features + J] = Cov[J*num_features + j] = accum / num_entries;
    }
}
```

```
dim3 blocks(SDIV(rows*cols,8), SDIV(rows*cols,8));
dim3 threads(8,8);
covariance_kernel<<<blocks,threads>>>(Data,Cov,imgs,rows*cols);
```

J = blockDim.x*blockIdx.x+threadIdx.x

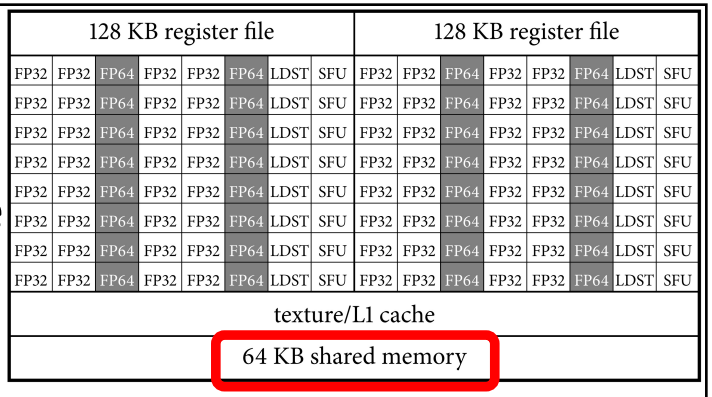
j = blockDim.y*blockIdx.y+threadIdx.y



CGMA (Compute-to-Global Memory Access) Ratio

- All threads access global memory for matrix elements
- 2 memory accesses (8 Bytes) per floating-point multiply-add;
- CGMA = 2 Flops / 8 Bytes
- P100: Global memory bandwidth ≈1 TB/s
- Thus, can load no more than We can expect no more than 1000/8 = **125 GFlop/s** for the naïve kernel
- FP32 Peak performance of a P100 is around 11 TFlop/s

Shared Memory in CUDA



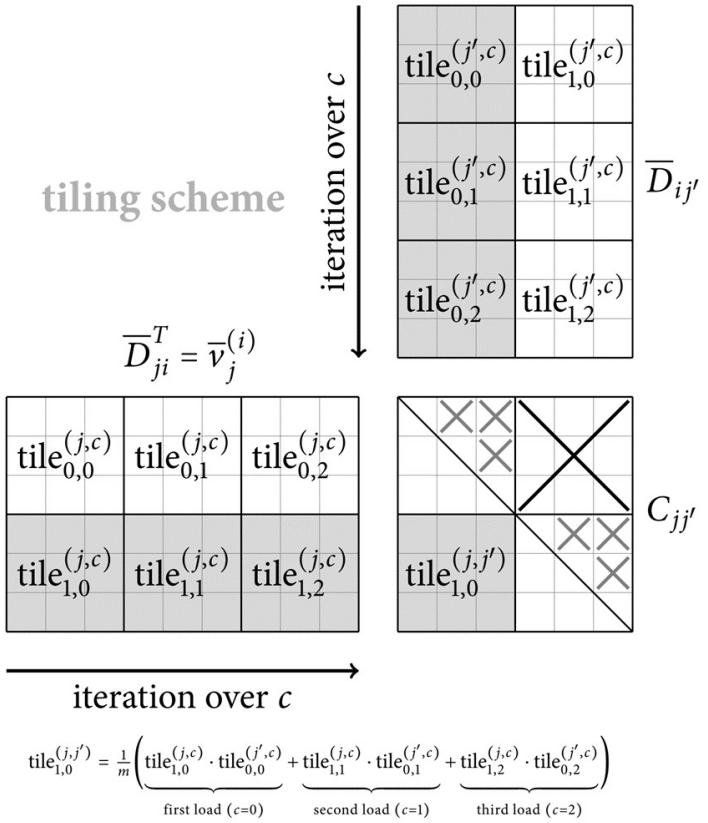
- Special type of memory whose contents explicitly declare
- Located within the SM (typically of size 64 KB per SM)
- Accessed at much higher speed (both latency and throughput)
- Still accessed by memory access instructions
- Commonly referred to as *scratchpad* memory

Variable declaration		Memory	Scope
	int LocalVar;	Register	Thread
__shared__	**int SharedVar;**	Shared	Block
__device__	int GlobalVar;	Global	Grid
__constant__	int ConstantVar;	Constant	Grid

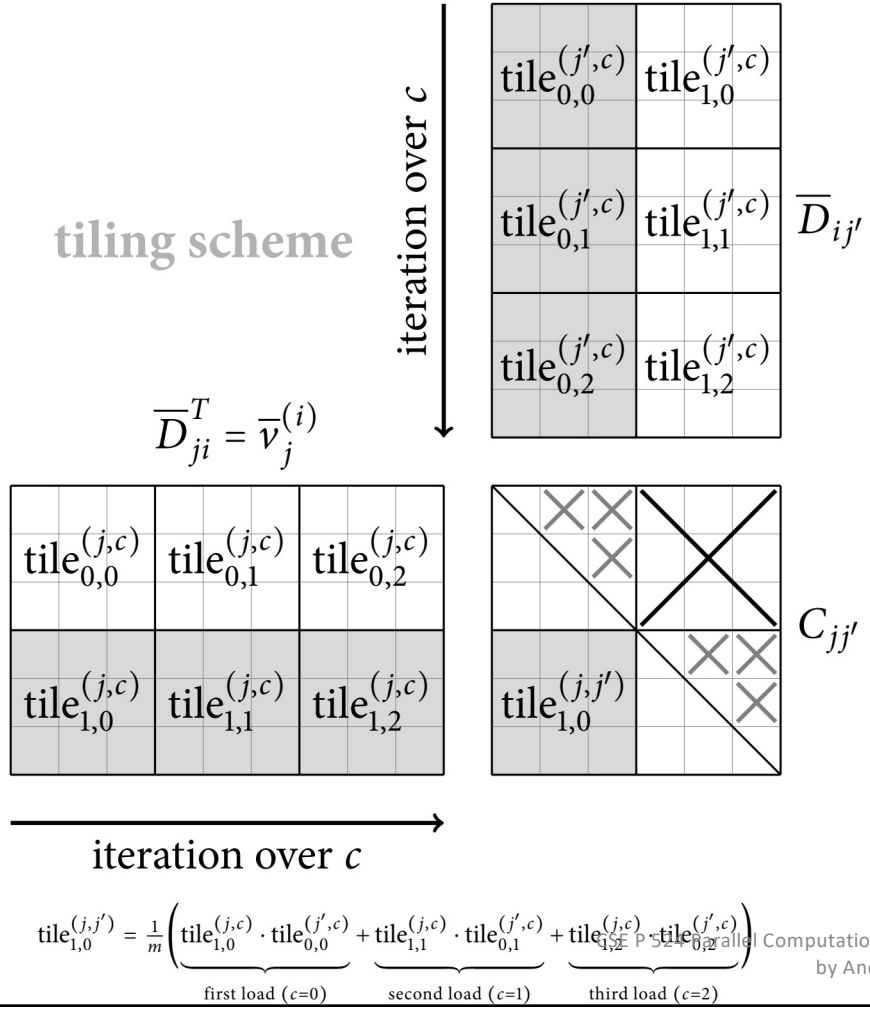
```
//allocation of shared memory in the CUDA kernel performing efficient
// covariance matrix computation
__shared__ value_t cache_x[chunk_size][chunk_size];
__shared__ value_t cache_y[chunk_size][chunk_size];
```

Shared Memory to reuse global memory data

- Each input element is read by ***n*** threads.
- Load each element into *Shared Memory* and have several threads use the local version to reduce the memory bandwidth ⇒ ***Tiled algorithms***

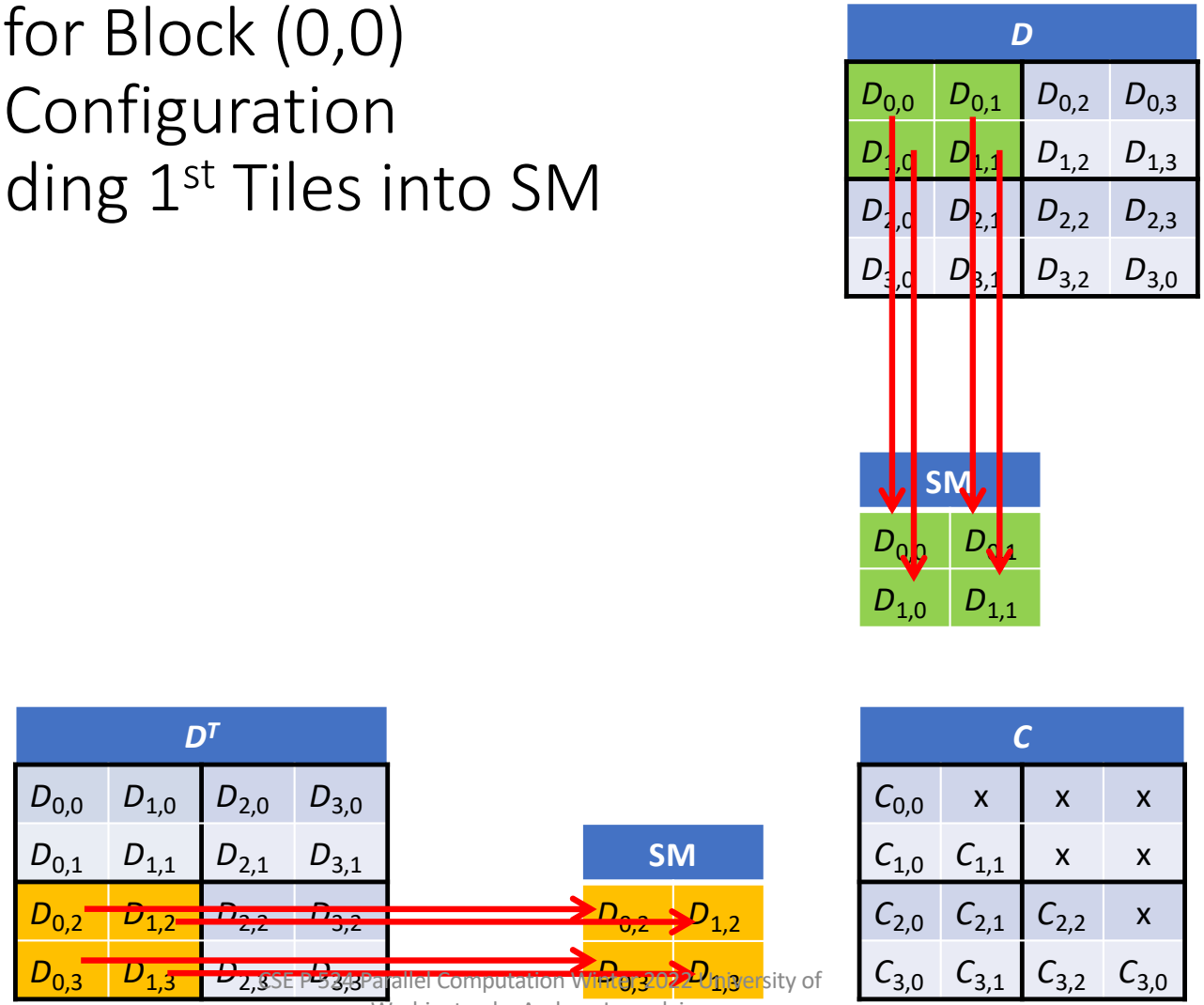


Tiled Covariance Computation

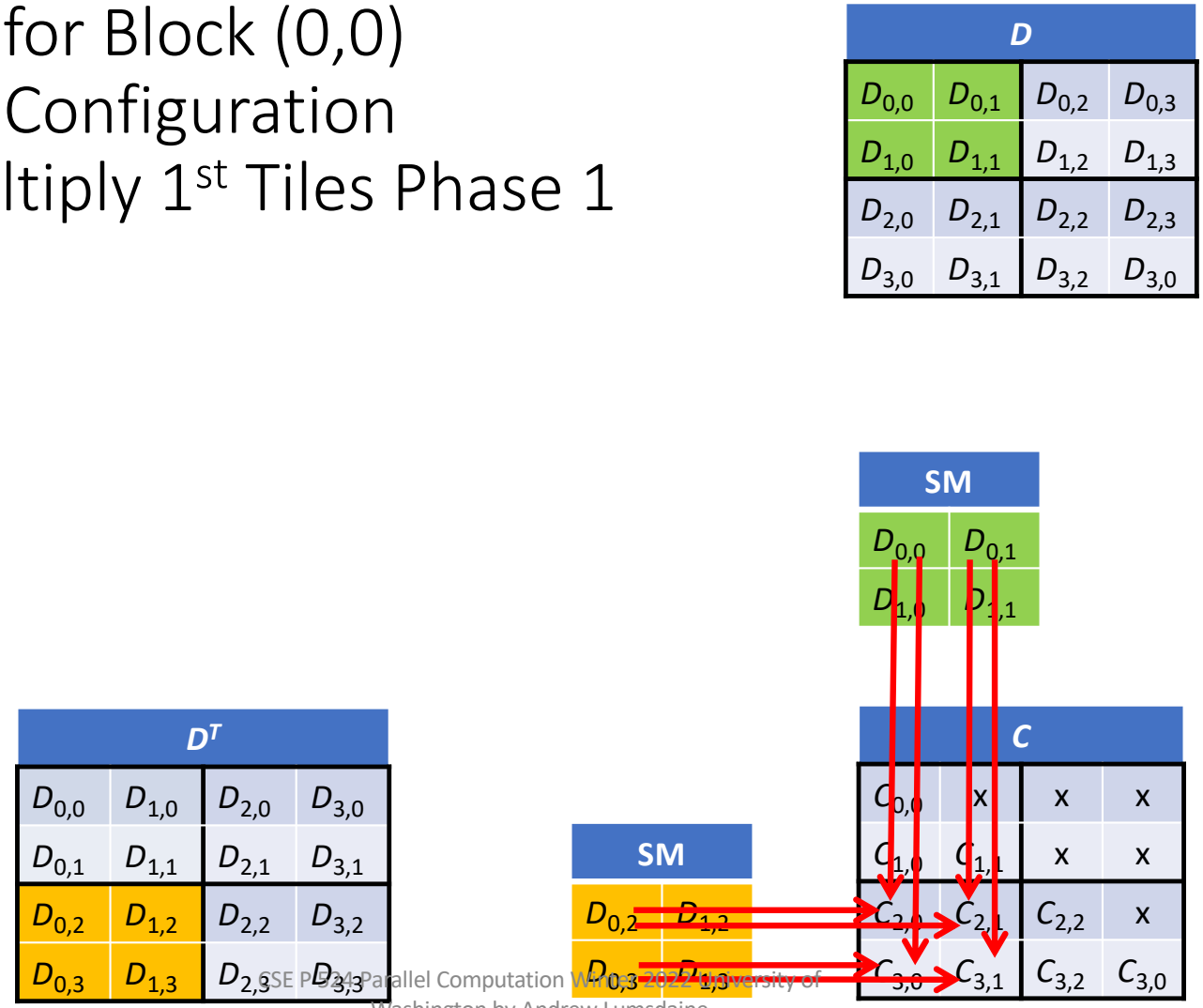


- **Idea:** Break up the execution of the kernel into phases so that the data accesses in each phase is focused on one subset (tile) of $\overline{D}$ and $\overline{D}^T$
- Example:
  - $m \times n = 9 \times 6$
  - $w \times w = 3 \times 3$
  - $m/w = 3$ iterations
- During each iteration:
  - Load into shared memory
  - Perform partial matrix product per tile by $w \times w$ threads
- As a result we can decrease accesses to global memory by a factor of w

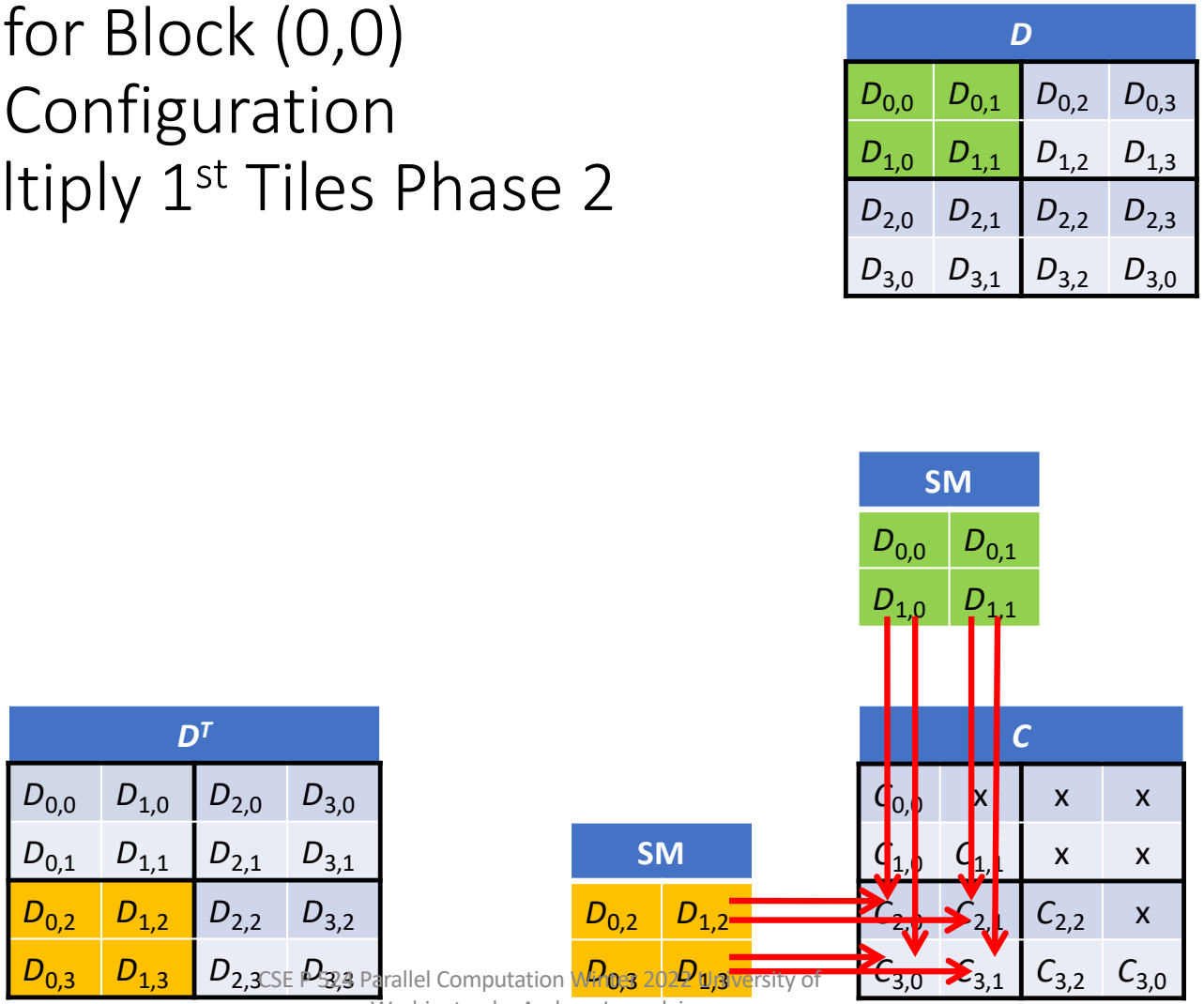
Work for Block (0,0)
w = 2 Configuration
1. Loading 1st Tiles into SM



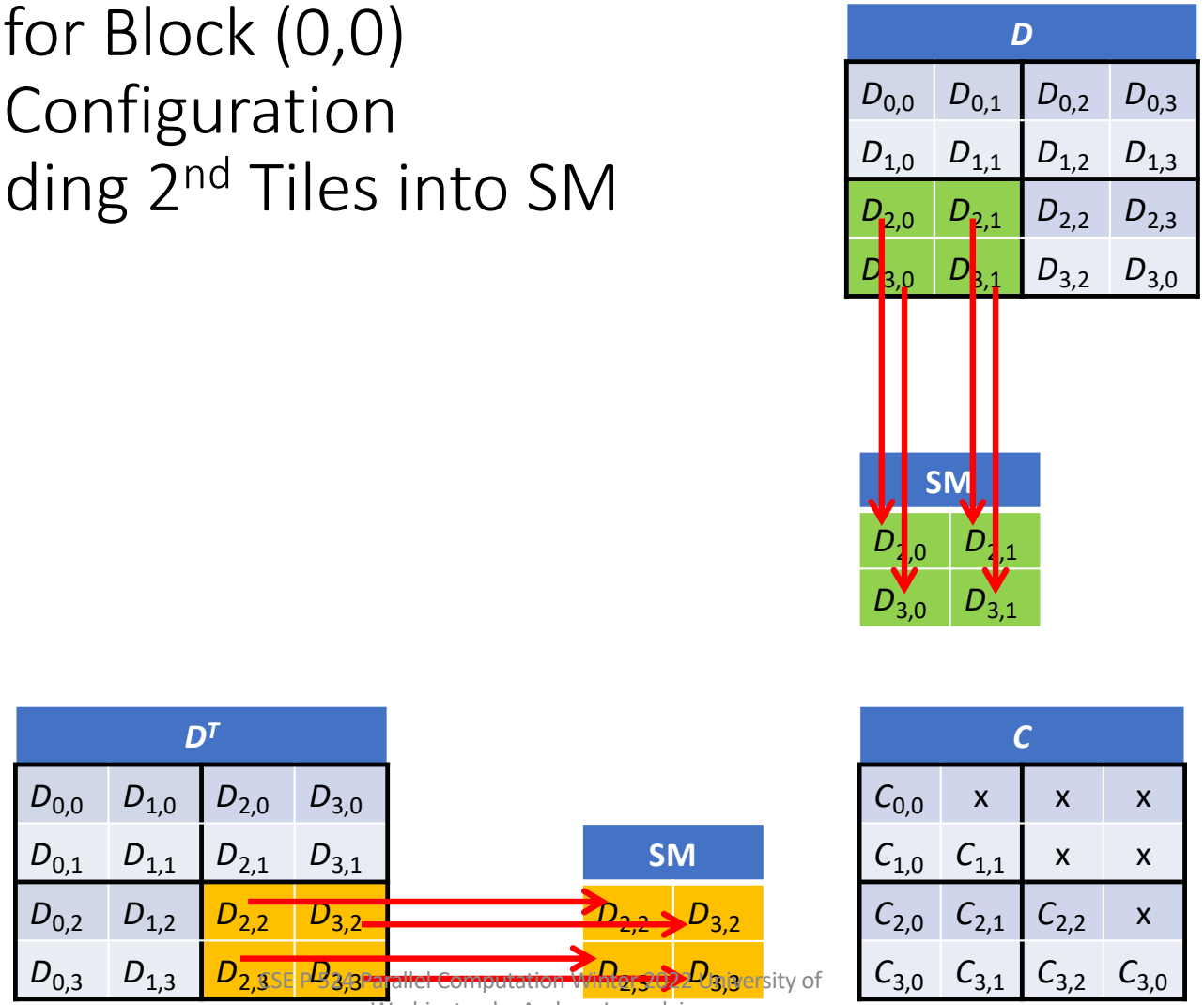
Work for Block (0,0)
w = 2 Configuration
2. Multiply 1st Tiles Phase 1



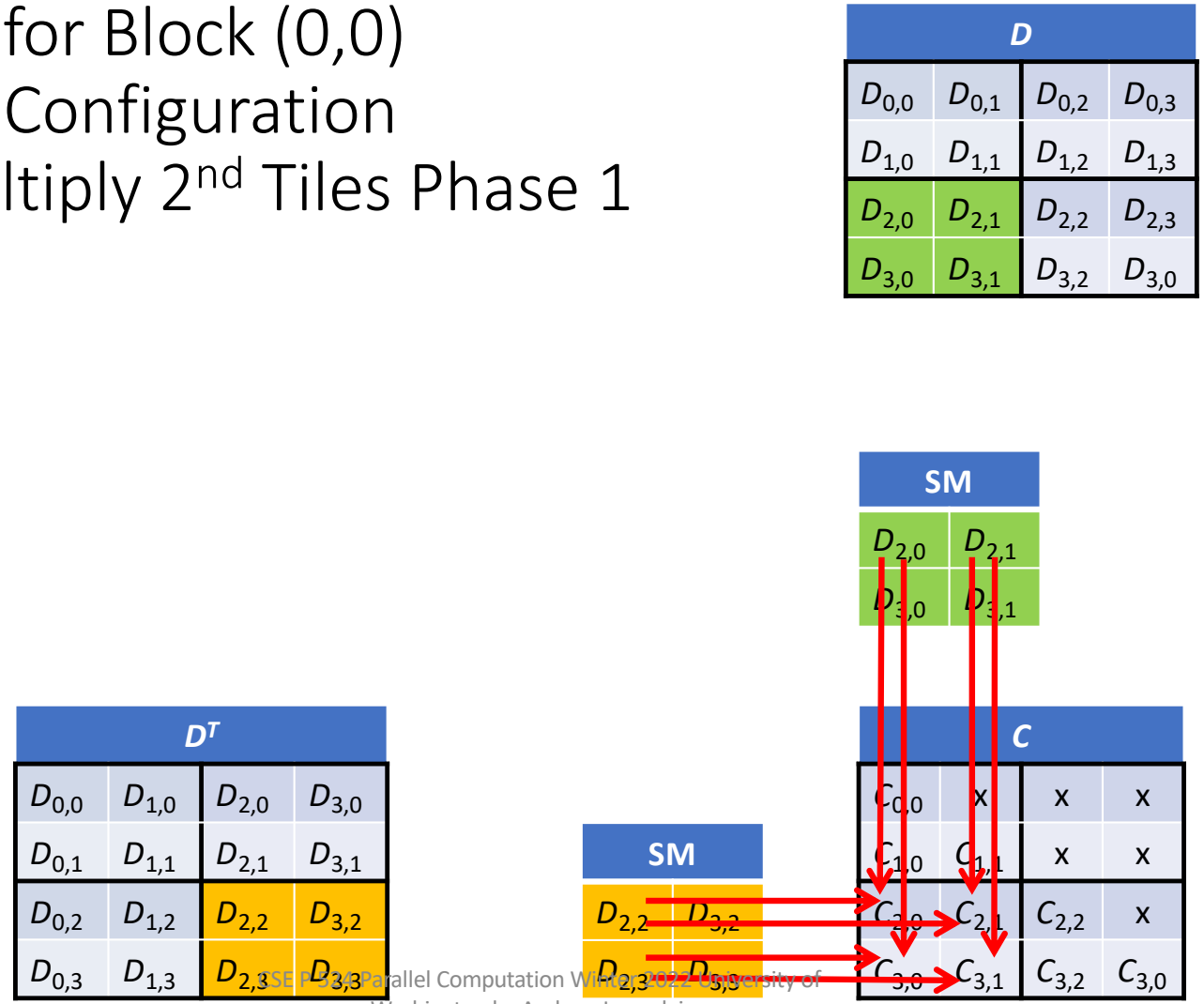
Work for Block (0,0)
w = 2 Configuration
3. Multiply 1st Tiles Phase 2



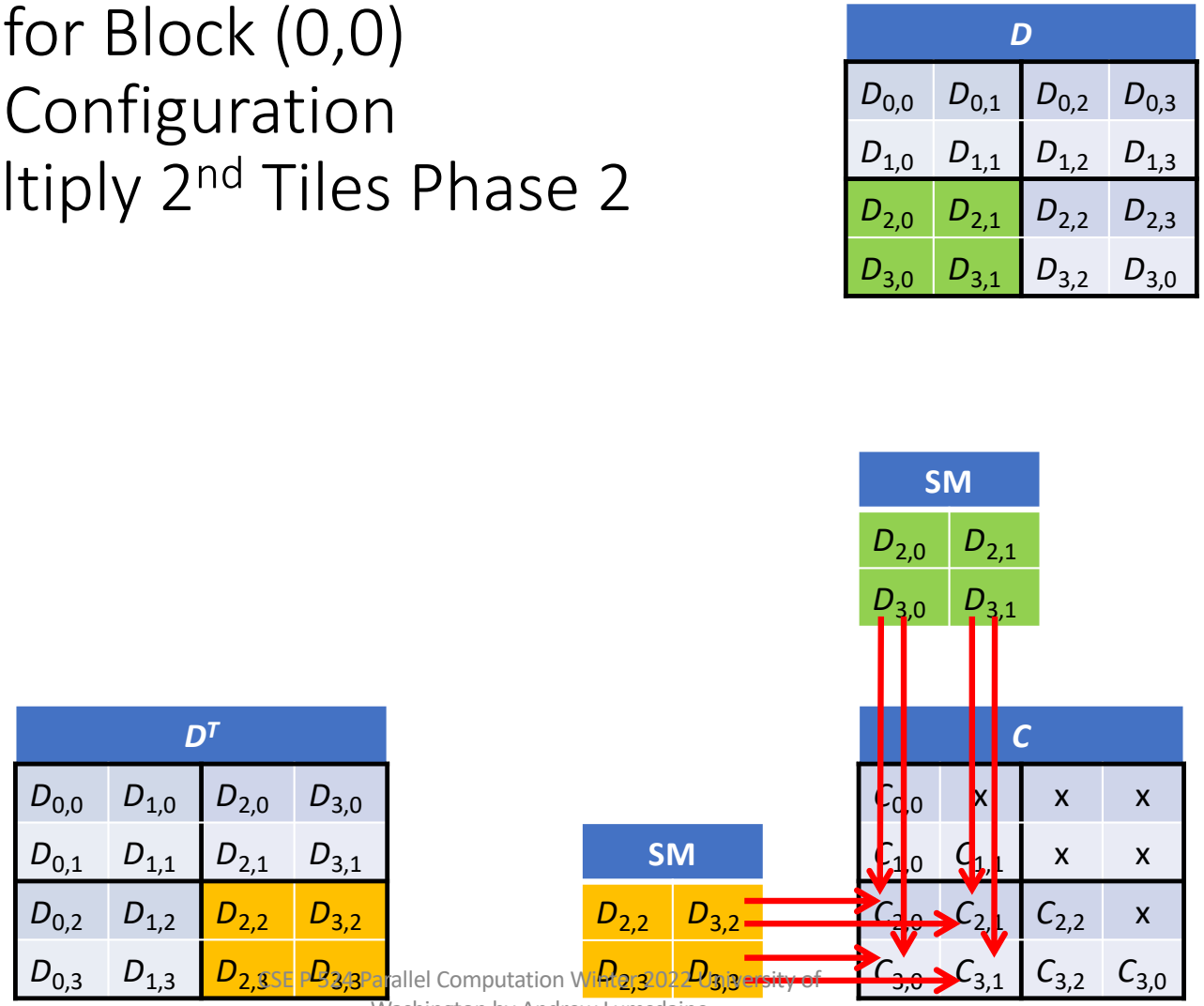
Work for Block (0,0)
w = 2 Configuration
4. Loading 2nd Tiles into SM



Work for Block (0,0)
w = 2 Configuration
5. Multiply 2nd Tiles Phase 1



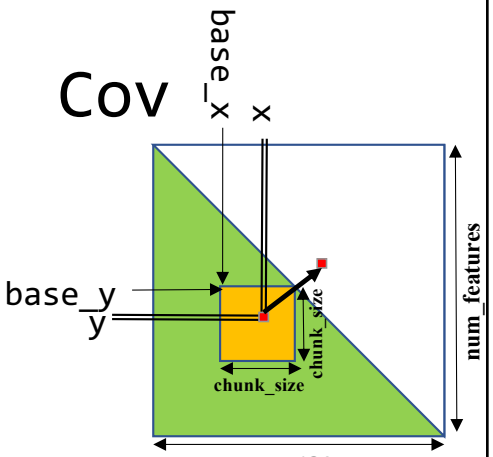
Work for Block (0,0)
w = 2 Configuration
5. Multiply 2nd Tiles Phase 2



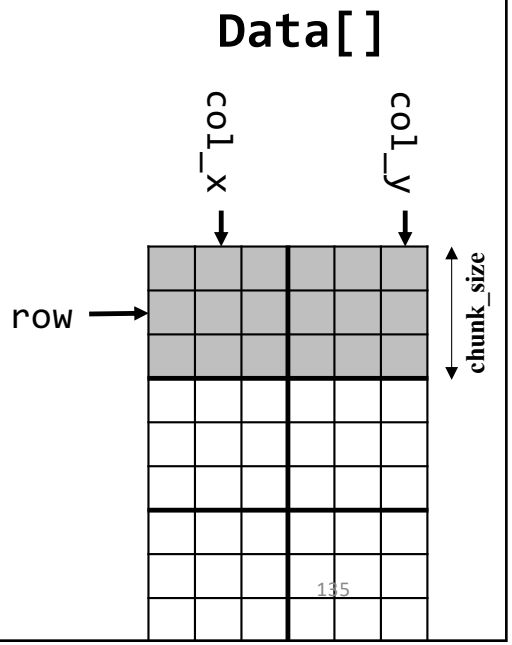
CUDA Cov Matrix

- Write a **CUDA** program that performs **tiled** covariance matrix computation with multiple thread blocks using **shared memory**

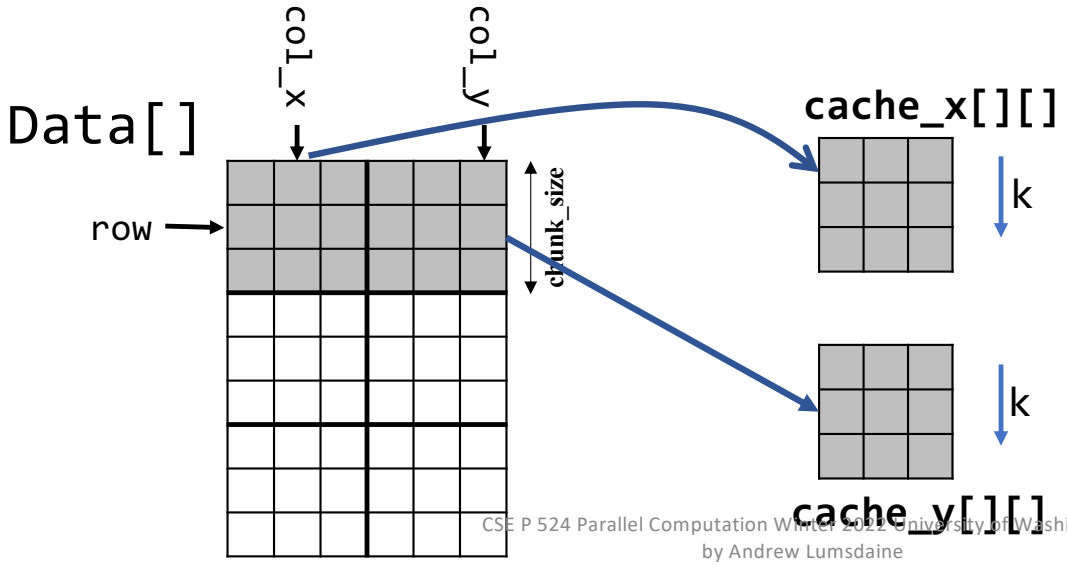
```
// Initial part of CUDA kernel performing efficient covariance
// matrix computation
template <typename index_t, typename value_t, uint32_t chunk_size = 8 >
__global__ void shared_covariance_kernel(value_t * Data, value_t * Cov,
index_t num_entries, index_t num_features) {
    // first index in a window of width chunk_size
    const index_t base_x = blockIdx.x*chunk_size;
    const index_t base_y = blockIdx.y*chunk_size;

    // local thread identifiers
    const index_t thid_y = threadIdx.y;
    const index_t thid_x = threadIdx.x;

    // global thread identifiers
    const index_t x = base_x + thid_x;
    const index_t y = base_y + thid_y;

    // optional early exit for tiles above the diagonal
    if (base_x > base_y) return;
```

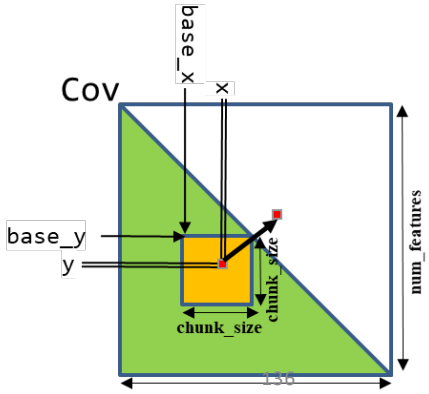


```
// allocate shared memory
__shared__ value_t cache_x[chunk_size][chunk_size];
__shared__ value_t cache_y[chunk_size][chunk_size];

// compute the number of chunks to be computed
const index_t num_chunks = SDIV(num_entries, chunk_size);

value_t accum = 0; // accumulated value of scalar product

//Start of the main loop for each chunk
for (index_t chunk = 0; chunk < num_chunks; chunk++) {
    // assign thread IDs to rows and columns
    const index_t row = thid_y + chunk*chunk_size;
    const index_t col_x = thid_x + base_x;
    const index_t col_y = thid_x + base_y;

    // check if valid row or column indices
    const bool valid_row = row < num_entries;
    const bool valid_col_x = col_x < num_features;
    const bool valid_col_y = col_y < num_features;
```



```
    // fill shared memory with tiles where thid_y enumerates image identifiers (entries) and
    // thid_x denotes feature coordinates (pixels). cache_x corresponds to x and cache_y to y
    cache_x[thid_y][thid_x] = valid_row*valid_col_x ? Data[row*num_features + col_x] : 0;
    cache_y[thid_y][thid_x] = valid_row*valid_col_y ? Data[row*num_features + col_y] : 0;
    __syncthreads(); // ensure that all threads have finished writing to shared memory

    if (x <= y) // optional early exit
        for (index_t k = 0; k < chunk_size; k++) // here we actually evaluate the scalar product
            accum += cache_y[k][thid_y] * cache_x[k][thid_x];
    __syncthreads(); // ensure that shared memory can safely be overwritten in the next iteration
} // end for loop over chunk entries
if (y < num_features && x <= y) Cov[y*num_features+x] = Cov[x*num_features+y] = accum / num_entries;
}
```





Performance Considerations

- chunk_size = 2
  - Each block has 2×2 = 4 threads
  - Each block performs 2×4 = 8 float loads from global memory for 4×(2×2) = 16 mul/add operations ⇒ **CGMA = 2**
- chunk_size = 8
  - Each block has 8×8 = 64 threads
  - Each block performs 2×64 = 128 float loads from global memory for 64×(2×8) = 1,024 mul/add operations ⇒ **CGMA = 8**
- Runtimes on a Titan X:
  - shared_covariance_kernel (chunk_size = 8): **980 ms**
  - Symmetrized naïve kernel: **18 sec**
  - **Speedup:** **18.4**
- Window size *w* is limited by the amount of available shared memory

Vectors

```
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>

#include <iostream>

int main(void) {
  thrust::host_vector<int> H(4);

  H[0] = 14;  H[1] = 20;  H[2] = 38;  H[3] = 46;

  std::cout << "H has size " << H.size() << std::endl;

  for (size_t i = 0; i < H.size(); i++)
    std::cout << "H[" << i << "] = " << H[i] << std::endl;

  H.resize(2);

  std::cout << "H now has size " << H.size() << std::endl;

  thrust::device_vector<int> D = H;

  D[0] = 99;  D[1] = 88;

  for (size_t i = 0; i < D.size(); i++)
    std::cout << "D[" << i << "] = " << D[i] << std::endl;

  return 0;
}
```

Vectors

Headers

```
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>

#include <iostream>

thrust::host_vector<int> H(4);

H[0] = 14;  H[1] = 20;  H[2] = 38;  H[3] = 46;

std::cout << "H has size " << H.size() << std::endl;

for (size_t i = 0; i < H.size(); i++)
  std::cout << "H[" << i << "] = " << H[i] << std::endl;

H.resize(2);

std::cout << "H now has size " << H.size() << std::endl;

thrust::device_vector<int> D = H;

D[0] = 99;  D[1] = 88;

for (size_t i = 0; i < D.size(); i++)
  std::cout << "D[" << i << "] = " << D[i] << std::endl;

return 0;
}
```

Vectors

Vector on host

```
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>

#include <iostream>

int main(void) {
 int main(void) {
   thrust::host_vector<int> H(4);

   H[0] = 14;  H[1] = 20;  H[2] = 38;  H[3] = 46;

   std::cout << "H has size " << H.size() << std::endl;

   for (size_t i = 0; i < H.size(); i++)
     std::cout << "H[" << i << "] = " << H[i] << std::endl;
  thrust::device_vector<int> D = H;

  D[0] = 99;  D[1] = 88;

  for (size_t i = 0; i < D.size(); i++)
    std::cout << "D[" << i << "] = " << D[i] << std::endl;

  return 0;
}
```

Vectors

```cpp
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>

#include <iostream>

int main(void) {
  thrust::host_vector<int> H(4);

  H[0] = 14;  H[1] = 20;  H[2] = 38;  H[3] = 46;

  std::cout << "H has size " << H.size() << std::endl;

  for (size_t i = 0; i < H.size(); i++)
    std::cout << "H[" << i << "] = " << H[i] << std::endl;

  H.resize(2);

  std::cout << "H now has size " << H.size() << std::endl;

  return 0;
}
```

Vectors

Declare host_vector H with 4 elements

Initialize elements

Print contents of H

```
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>

#include <iostream>

int main(void) {
```

```
int main(void) {
  thrust::host_vector<int> H(4);

  H[0] = 14;  H[1] = 20;  H[2] = 38;  H[3] = 46;

  std::cout << "H has size " << H.size() << std::endl;

  for (size_t i = 0; i < H.size(); i++)
    std::cout << "H[" << i << "] = " << H[i] << std::endl;
```

```
  thrust::device_vector<int> D = H;

  D[0] = 99;  D[1] = 88;

  for (size_t i = 0; i < D.size(); i++)
    std::cout << "D[" << i << "] = " << D[i] << std::endl;

  return 0;
}
```

Vectors

Copy host_vector H to device_vector D

Elements of D can be modified

Print contents of D

RAII

```
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>

#include <iostream>

int main(void) {
  thrust::host_vector<int> H(4);

  H[0] = 14;  H[1] = 20;  H[2] = 38;  H[3] = 46;

  std::cout << "H has size " << H.size() << std::endl;

  for (size_t i = 0; i < H.size(); i++)
    std::cout << "H[" << i << "] = " << H[i] << std::endl;

  H.resize(2);

  thrust::device_vector<int> D = H;

  D[0] = 99;  D[1] = 88;

  for (size_t i = 0; i < D.size(); i++)
    std::cout << "D[" << i << "] = " << D[i] << std::endl;

  return 0;
}
```

Vectors

Create vector on device

Initialize all elements to 10

Set first seven elements to 9

Initialize H with first 5 elements of D

Copy of all H back to beginning of D

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/fill.h>
#include <thrust/sequence.h>

int main() {

  thrust::device_vector<int> D(10, 1);

  thrust::fill(D.begin(), D.begin() + 7, 9);

  thrust::host_vector<int> H(D.begin(), D.begin() + 5);

  thrust::sequence(H.begin(), H.end());

  thrust::copy(H.begin(), H.end(), D.begin());

  for (size_t i = 0; i < D.size(); i++)
    std::cout << "D[" << i << "] = " << D[i] << std::endl;

  return 0;
```

Iterator (device)

Iterator (host)

Genericity!

SL compatibility

Genericity!

Genericity!

```
#include <thrust/device_vector.h>
#include <thrust/copy.h>
#include <list>
#include <vector>

int main(void)
{
  std::list<int> stl_list;

  stl_list.push_back(10);
  stl_list.push_back(20);
  stl_list.push_back(30);
  stl_list.push_back(40);

  thrust::device_vector<int> D(stl_list.begin(), stl_list.end());

  std::vector<int> stl_vector(D.size());
  thrust::copy(D.begin(), D.end(), stl_vector.begin());

  return 0;
}
```

Transformations

```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/sequence.h>
#include <thrust/copy.h>
#include <thrust/fill.h>
#include <thrust/replace.h>
#include <thrust/functional.h>

int main(void)
{
  thrust::device_vector<int> X(10);
  thrust::device_vector<int> Y(10);
  thrust::device_vector<int> Z(10);

  thrust::sequence(X.begin(), X.end());

  thrust::transform(X.begin(), X.end(), Y.begin(), thrust::negate<int>());

  thrust::fill(Z.begin(), Z.end(), 2);

  thrust::transform(X.begin(), X.end(), Z.begin(), Y.begin(), thrust::modulus<int>());

  thrust::replace(Y.begin(), Y.end(), 1, 10);

  thrust::copy(Y.begin(), Y.end(), std::ostream_iterator<int>(std::cout, "\n"));

  return 0;
}
```

Transformations

```
int main(void)
{
  thrust::device_vector<int> X(10);
  thrust::device_vector<int> Y(10);
  thrust::device_vector<int> Z(10);

  thrust::sequence(X.begin(), X.end());

  thrust::transform(X.begin(), X.end(), Y.begin(), thrust::negate<int>());

  thrust::fill(Z.begin(), Z.end(), 2);

  thrust::transform(X.begin(), X.end(), Z.begin(), Y.begin(), thrust::modulus<int>());

  thrust::replace(Y.begin(), Y.end(), 1, 10);

  thrust::copy(Y.begin(), Y.end(), std::ostream_iterator<int>(std::cout, "\n"));

  return 0;
}
```

saxpy

```
struct saxpy_functor : public thrust::binary_function<float, float, float> {
  saxpy_functor(float _a) : a(_a) {}

  __host__ __device__ float operator()(const float& x, const float& y) const { return a * x + y; }

  const float a;
};

void saxpy_fast(float A, thrust::device_vector<float>& X, thrust::device_vector<float>& Y) {
  thrust::transform(X.begin(), X.end(), Y.begin(), Y.begin(), saxpy_functor(A));
}

void saxpy_slow(float A, thrust::device_vector<float>& X, thrust::device_vector<float>& Y) {
  thrust::device_vector<float> temp(X.size());
  thrust::fill(temp.begin(), temp.end(), A);
  thrust::transform(X.begin(), X.end(), temp.begin(), temp.begin(), thrust::multiplies<float>());
  thrust::transform(temp.begin(), temp.end(), Y.begin(), Y.begin(), thrust::plus<float>());
```

Temp <- A

Temp <- A*X

Temp <- A*X + Y

Norm

operator()

norm!

```
template <typename T>
struct square
{
  __host__ __device__
  T operator()(const T& x) const { return x * x; }
};

int main(void)
{
  float x[4] = {1.0, 2.0, 3.0, 4.0};

  thrust::device_vector<float> d_x(x, x + 4);

  square<float>        unary_op;
  thrust::plus<float> binary_op;
  float init = 0;

  float norm = std::sqrt( thrust::transform_reduce(d_
  ↪  d_x.end(), unary_op, init, binary_op) );

  std::cout << norm << std::endl;

  return 0;
}
```

unary op

binary op

Fancy iterators

```
#include <thrust/iterator/transform_iterator.h>

thrust::device_vector<int> vec(3);
vec[0] = 10; vec[1] = 20; vec[2] = 30;

// create iterator (type omitted)
...
first = thrust::make_transform_iterator(vec.begin(), negate<int>());
...
last  = thrust::make_transform_iterator(vec.end(),   negate<int>());

first[0]   // returns -10
first[1]   // returns -20
first[2]   // returns -30

thrust::reduce(first, last);
```

transform_iterator

constant_iterator

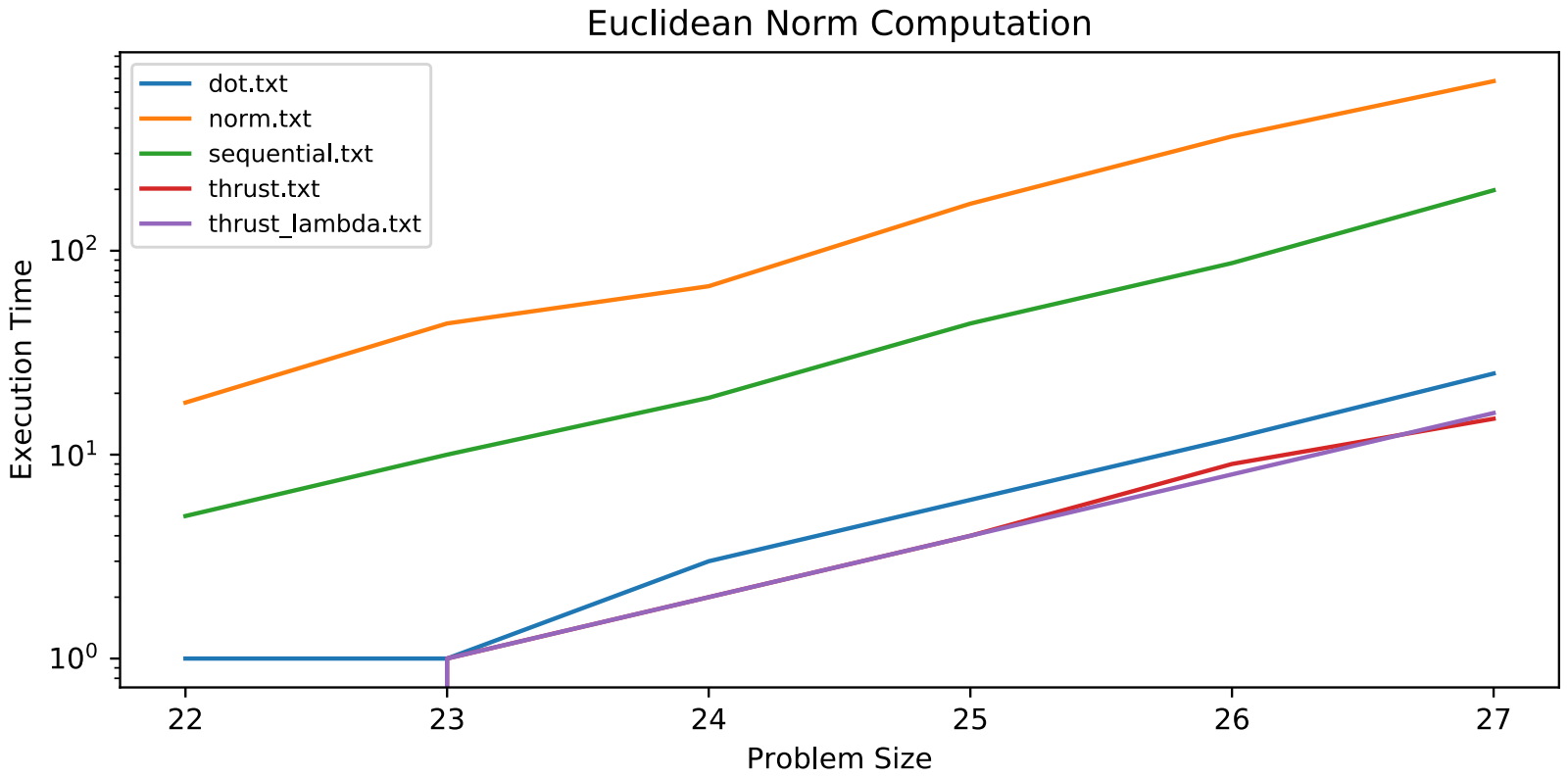
counting_iterator

zip_iterator

Execution policies

- Thrust allows GPU, CPU, OpenMP execution policies

Performance



Other GPU programming systems

- OpenACC
- OpenCL
- SyCL
- CuSP
- CuBLAS
- Halide
- Etc.

Thank you!

Creative Commons BY-NC-SA 4.0 License