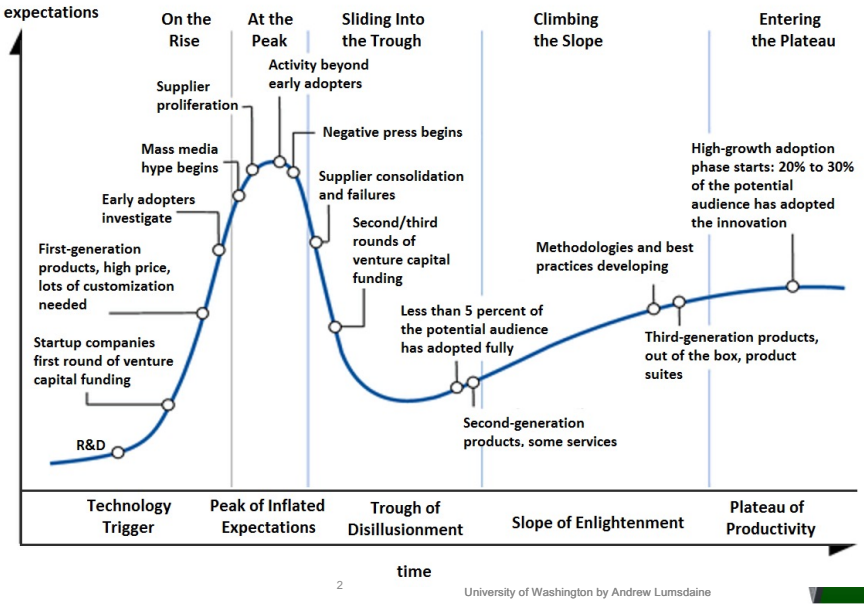
# AMATH 483/583
# High Performance Scientific Computing

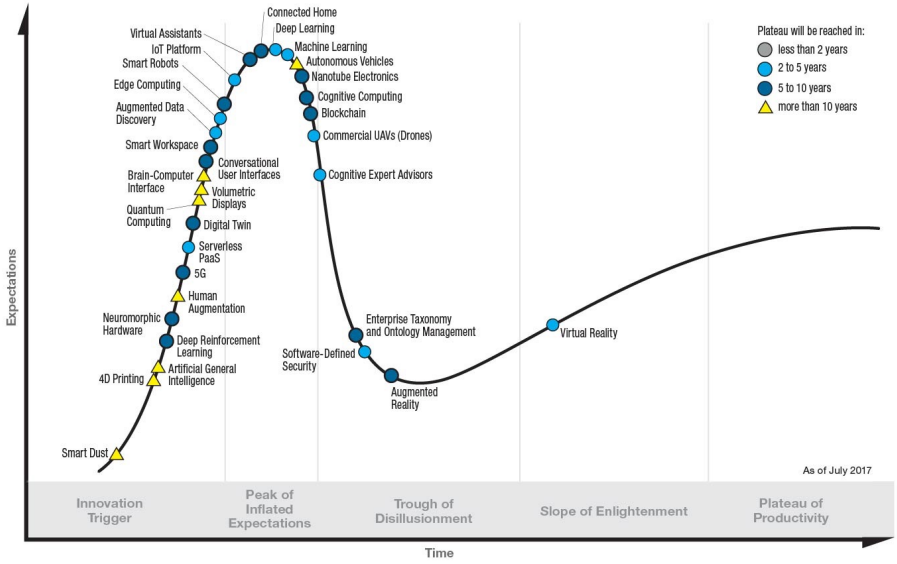
## Lecture 15
## GPUs, CUDA, Thrust

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

## The Hype Cycle

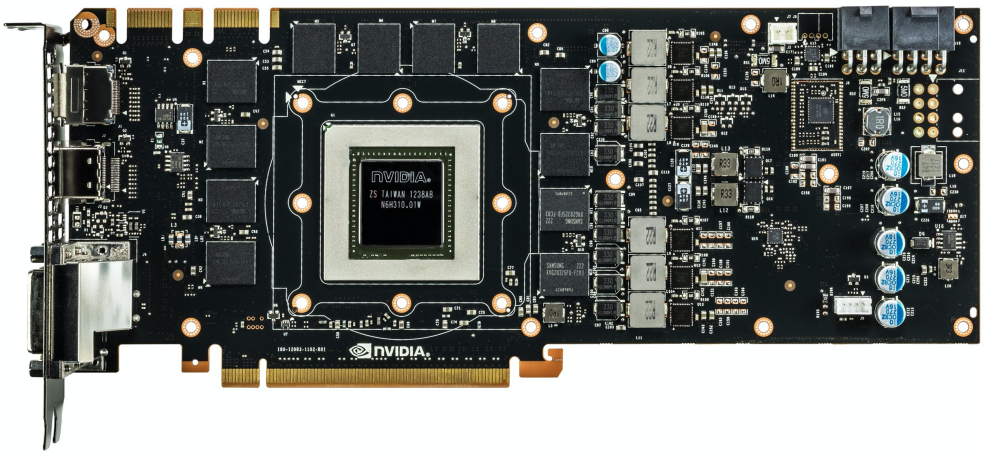


# Gartner Hype Cycle

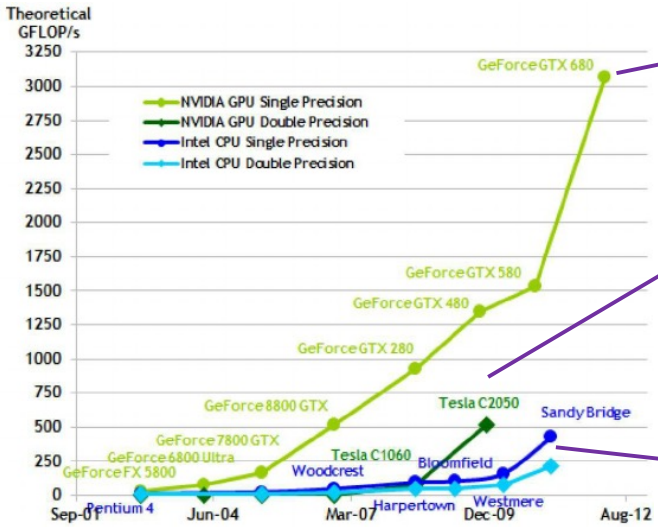





# GPU



# Performance




Explain

Explain

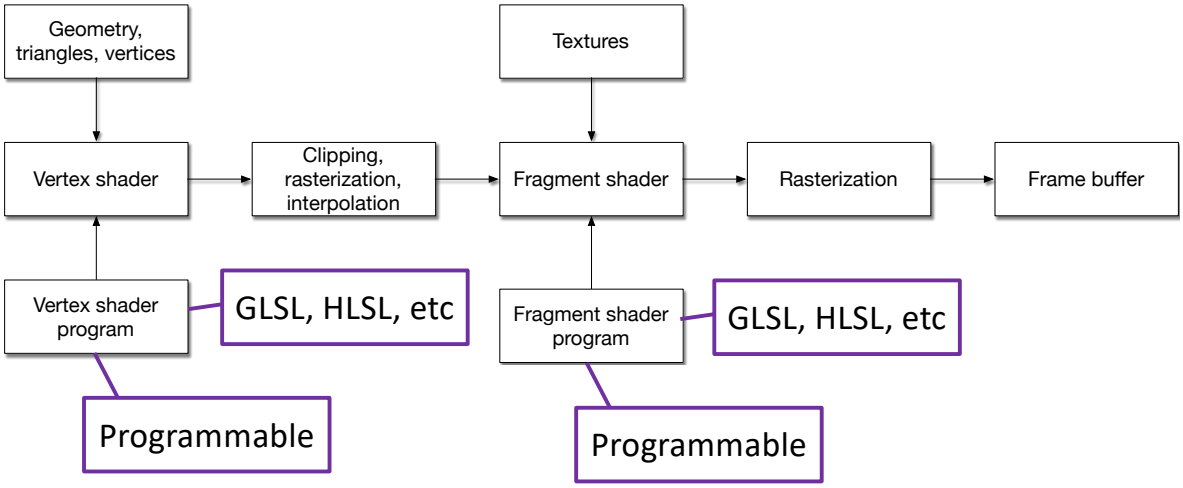
Explain



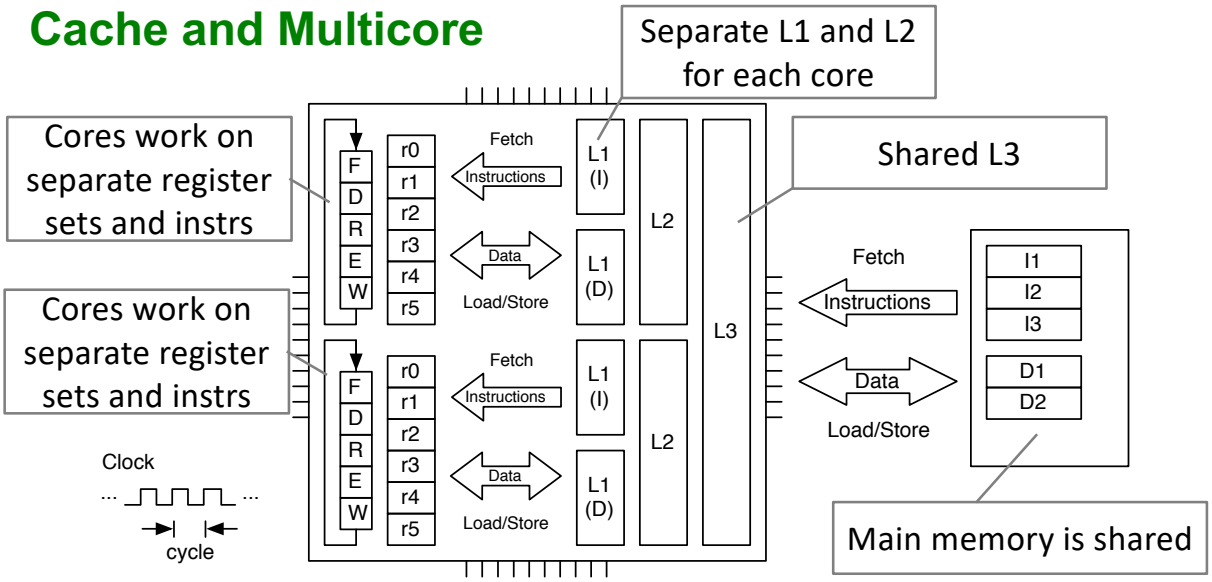
Harju, Ari & Siro, Topi & Canova, Filippo & Hakala, Samuli & Rantalaiho, Teemu. (2012). Computational Physics on Graphics Processing Units. 10.1007/978-3-642-36803-5_1.



# Graphics Pipeline

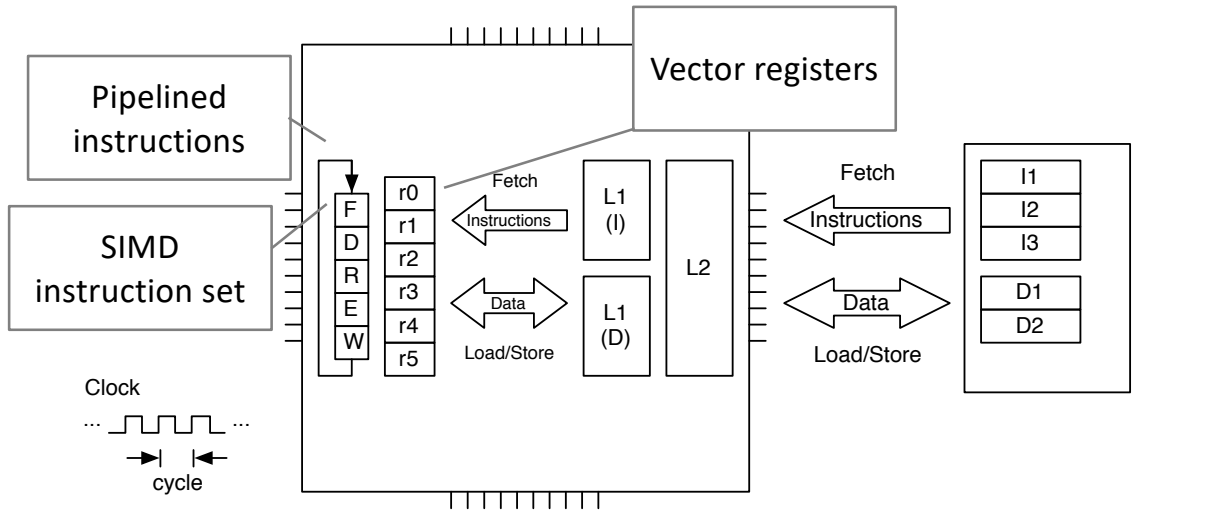


## Cache and Multicore

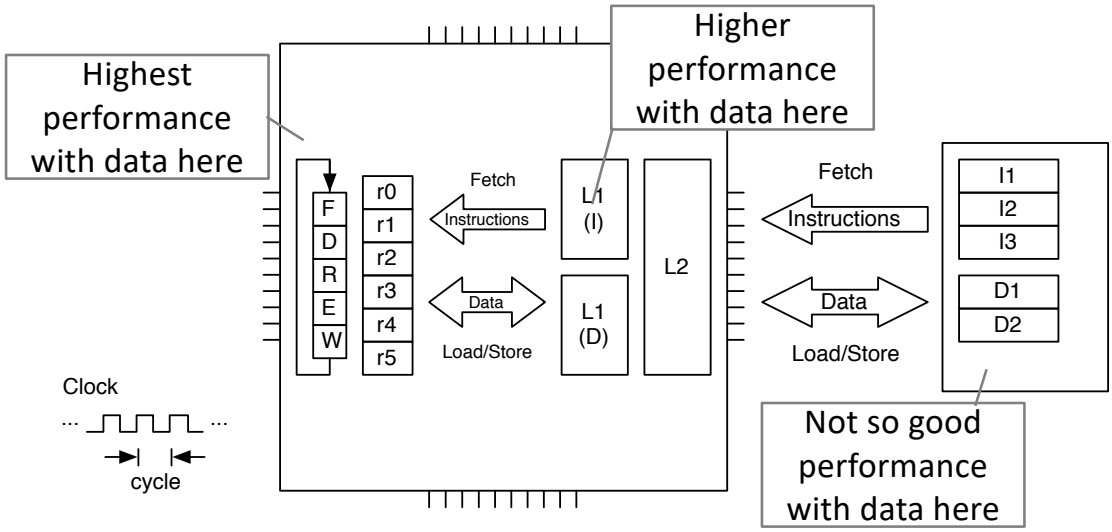





## Performance (Instruction-Level Parallelism)

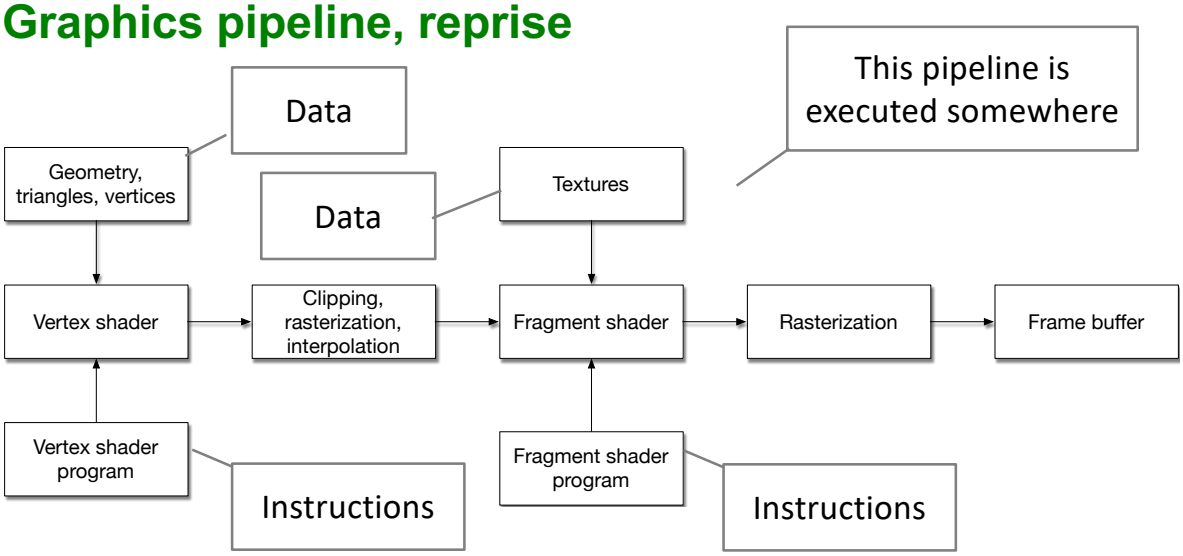


# Performance (Memory Hierarchy)

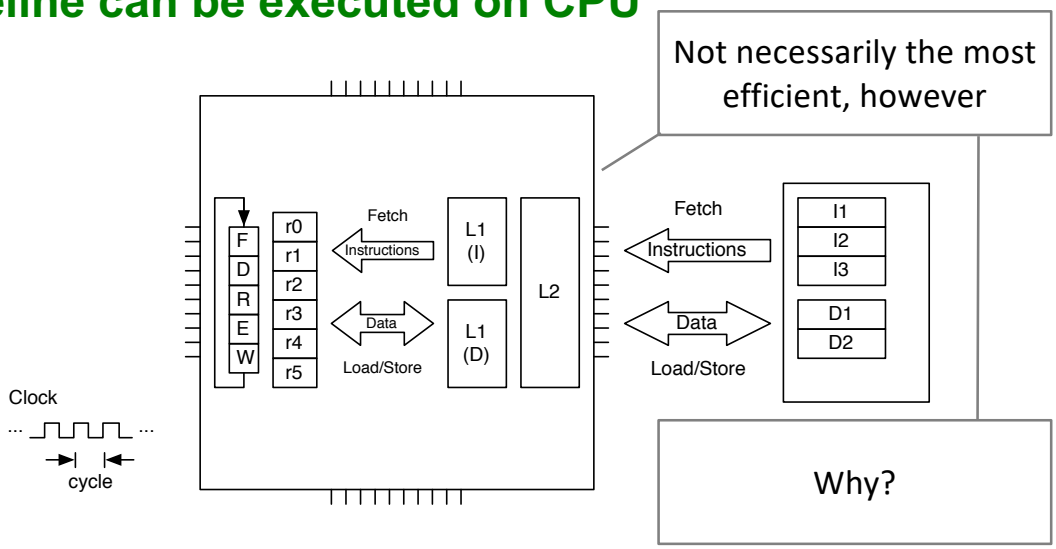





# Graphics pipeline, reprise



# Pipeline can be executed on CPU

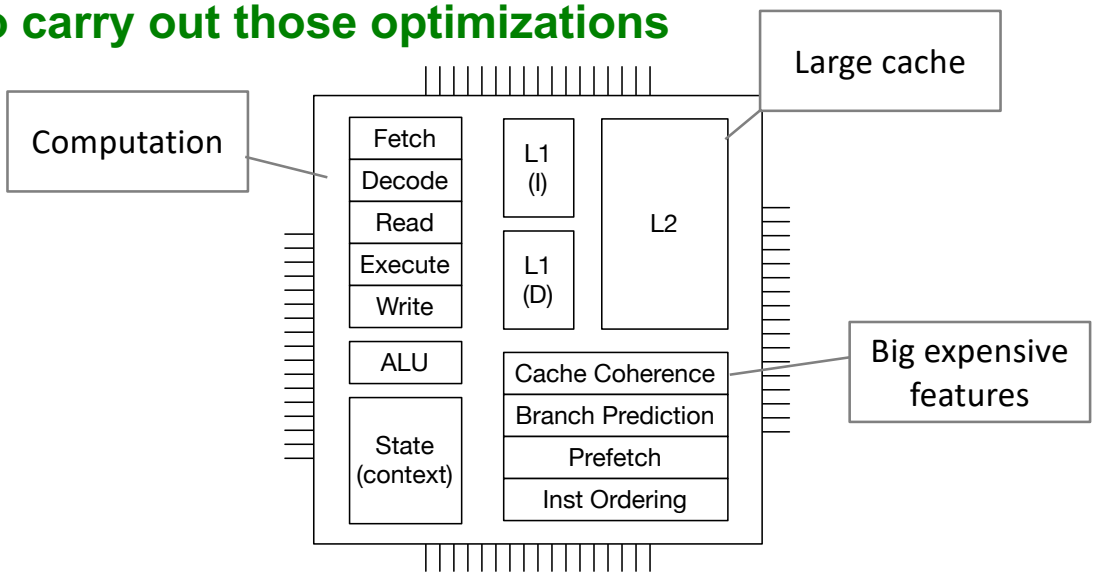





# For what are CPUs optimized?

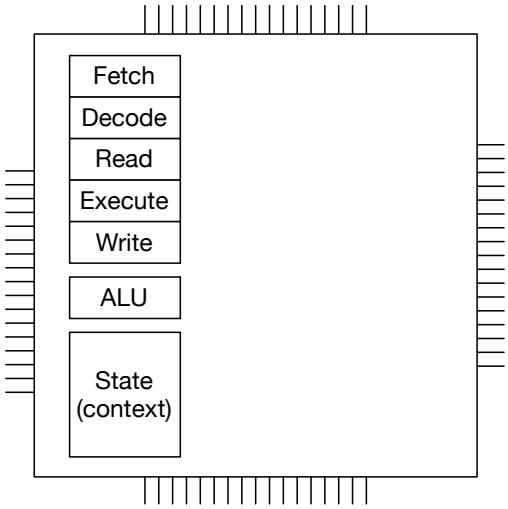
- CPUs are optimized for _____

## To carry out those optimizations

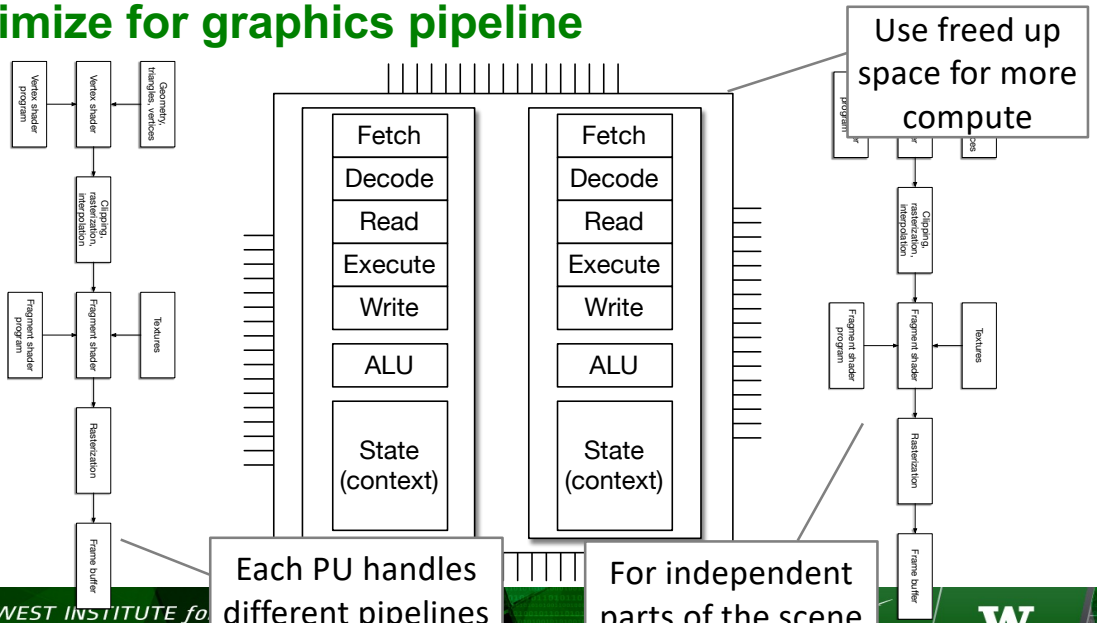





## Optimize instead for graphics pipeline

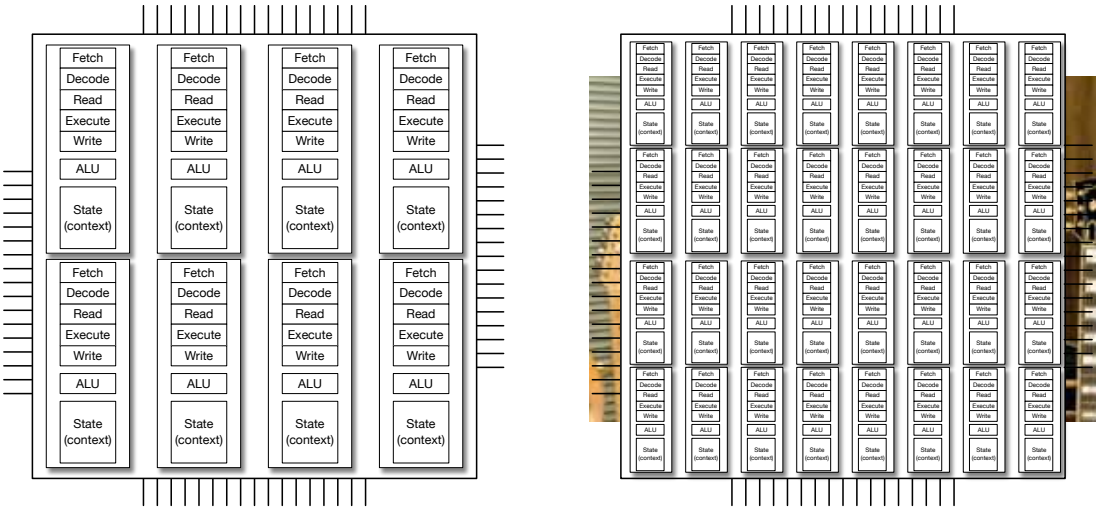


# Optimize for graphics pipeline

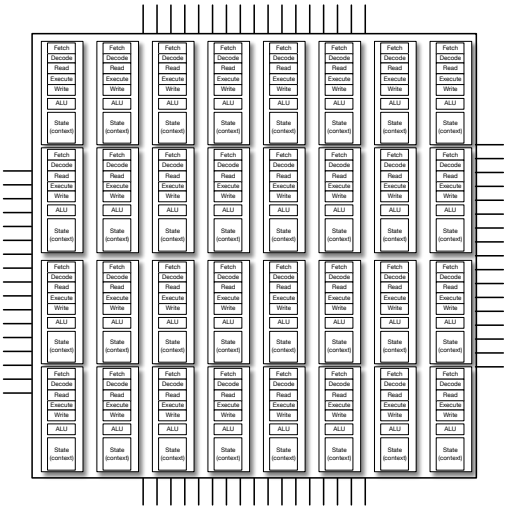





# Eight is better than two, innit?



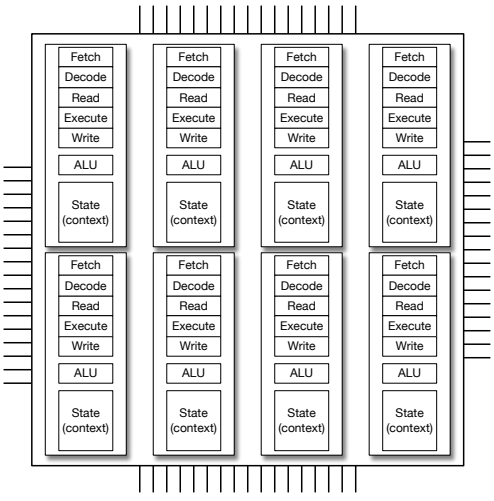
# But there is just one problem

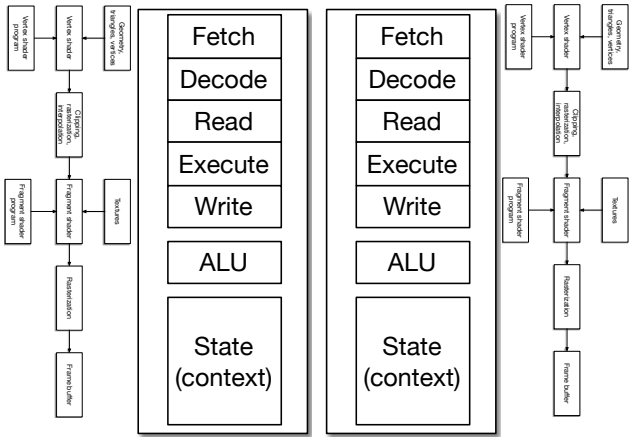



- And the problem is _____

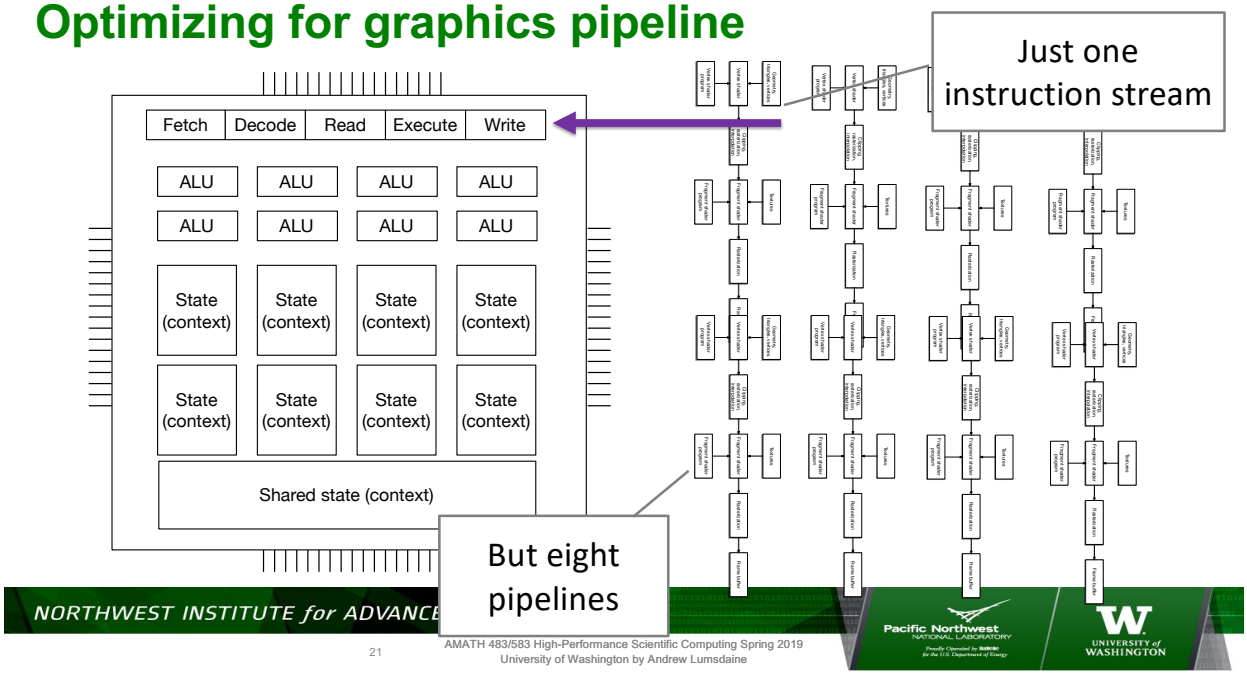


# Redundant hardware



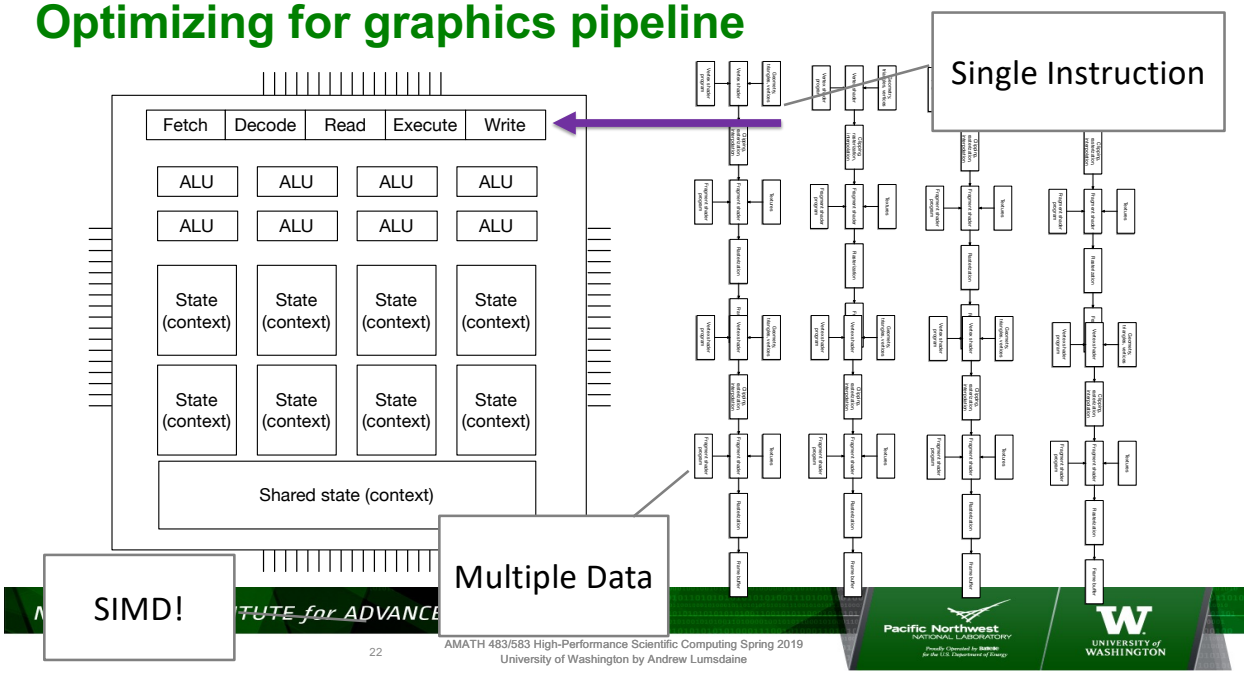



# Optimizing for graphics pipeline





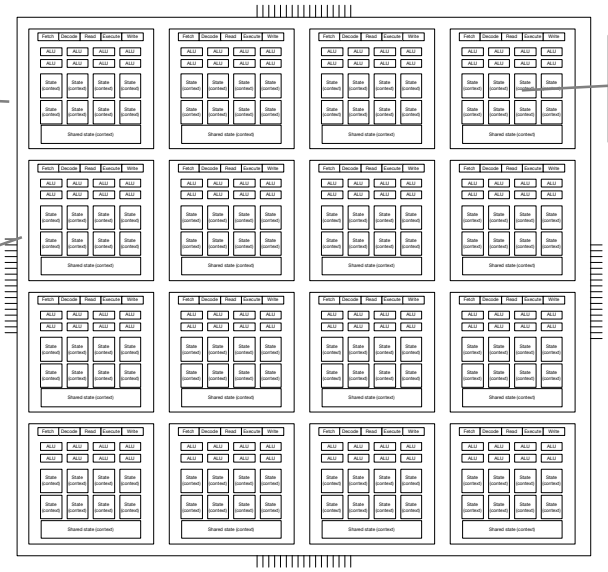

# Optimizing for graphics pipeline



# Why stop now

16 cores

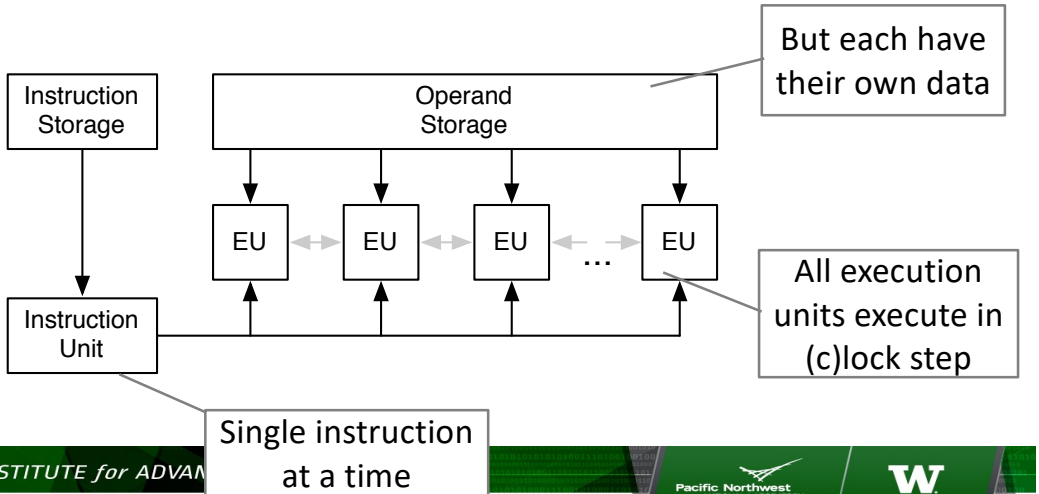
128 ALUs



16 instruction streams

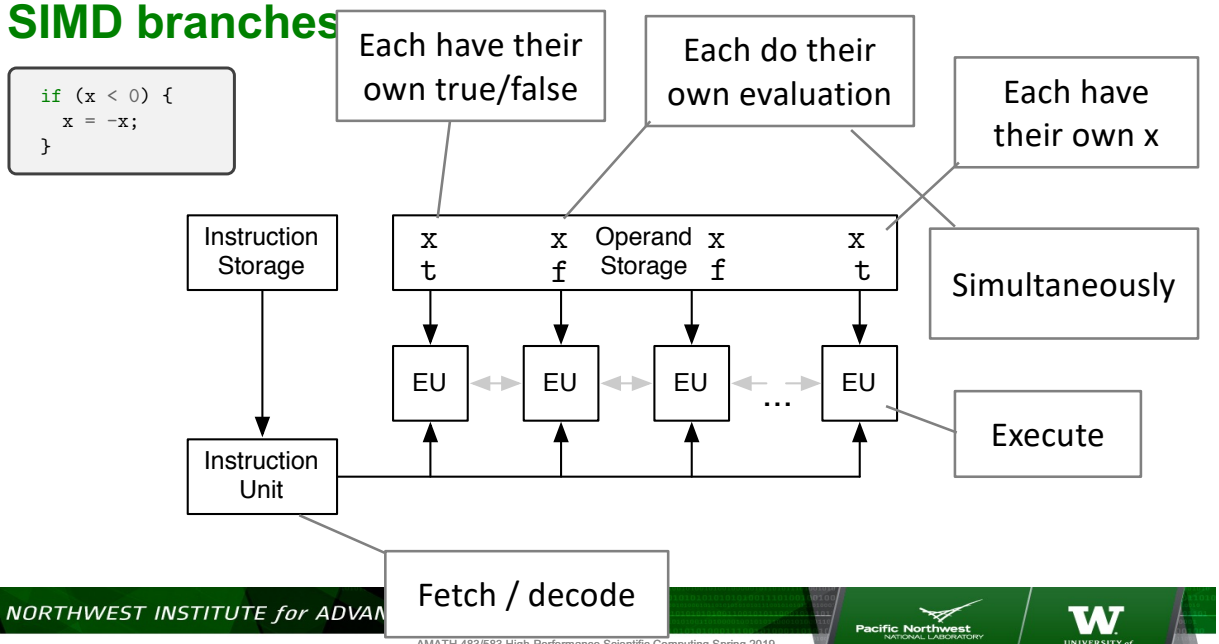


# SIMD branches



## SIMD branches

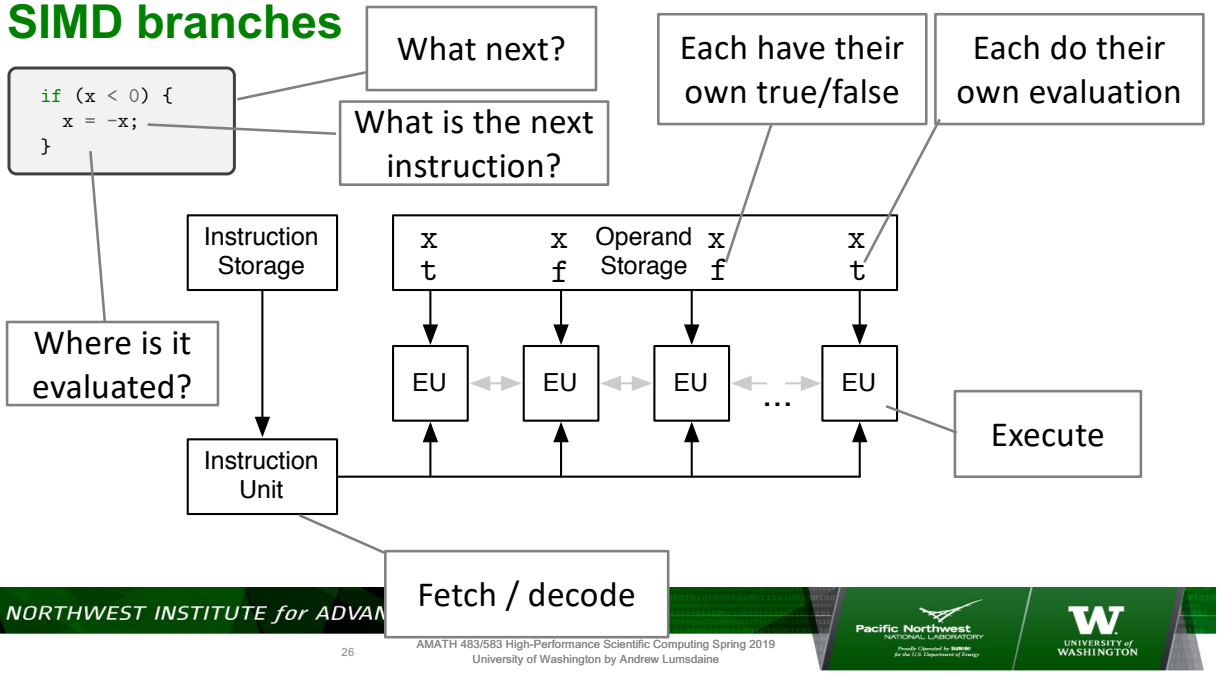
```
if (x < 0) {
  x = -x;
}
```





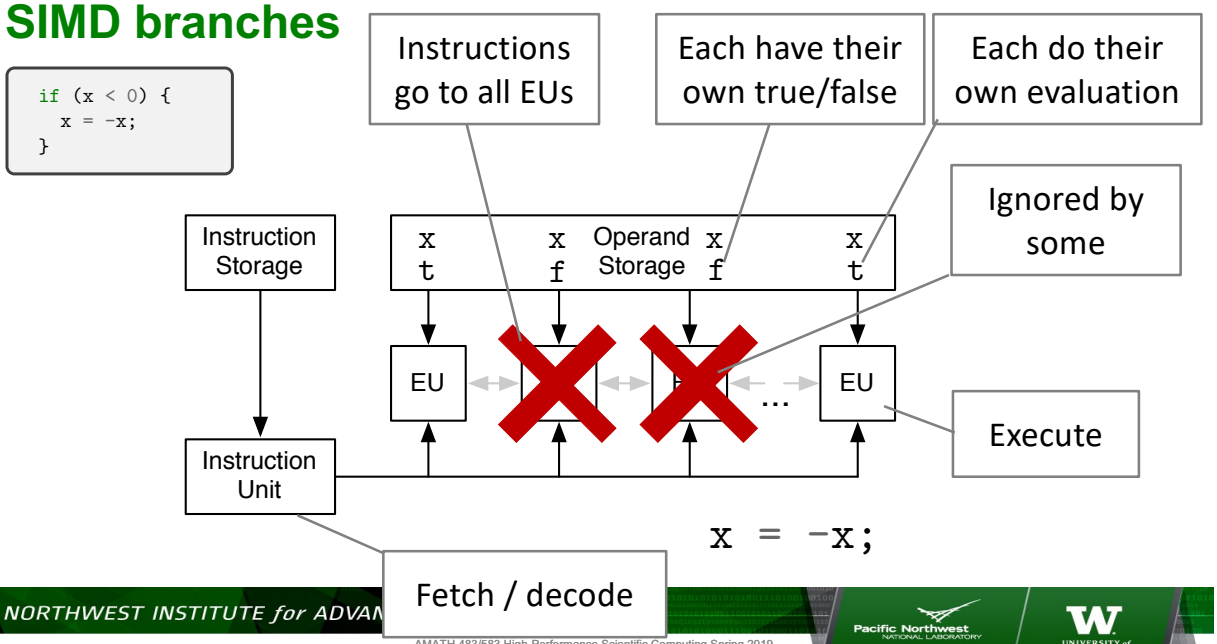

## SIMD branches

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



# SIMD branches

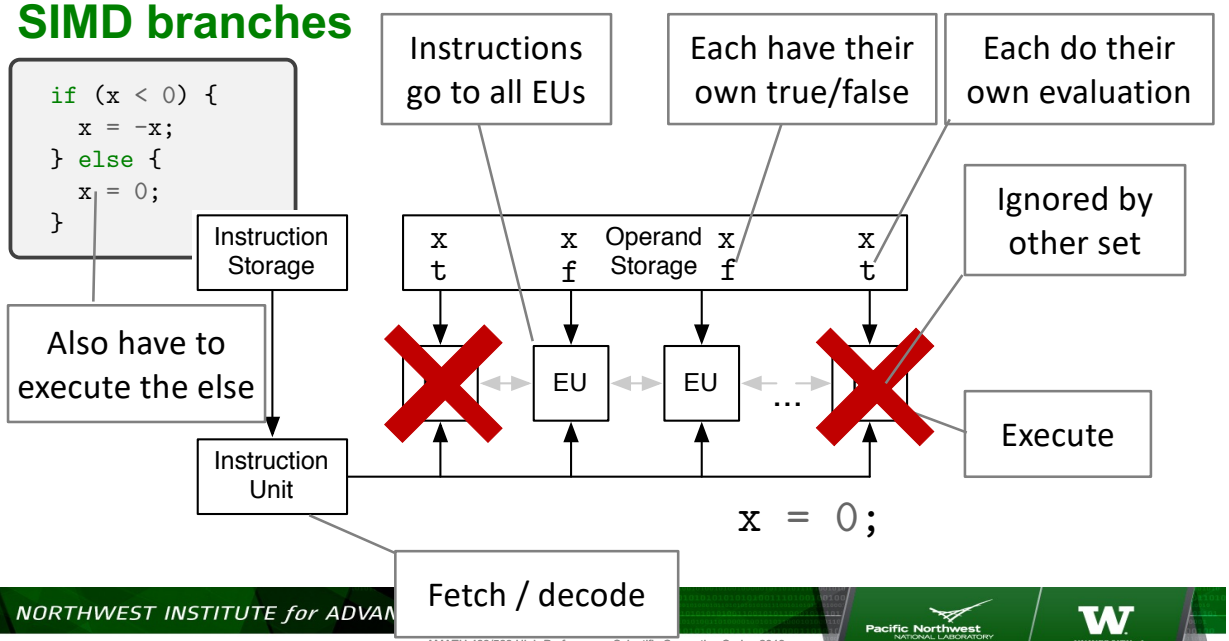
```
if (x < 0) {
  x = -x;
}
```



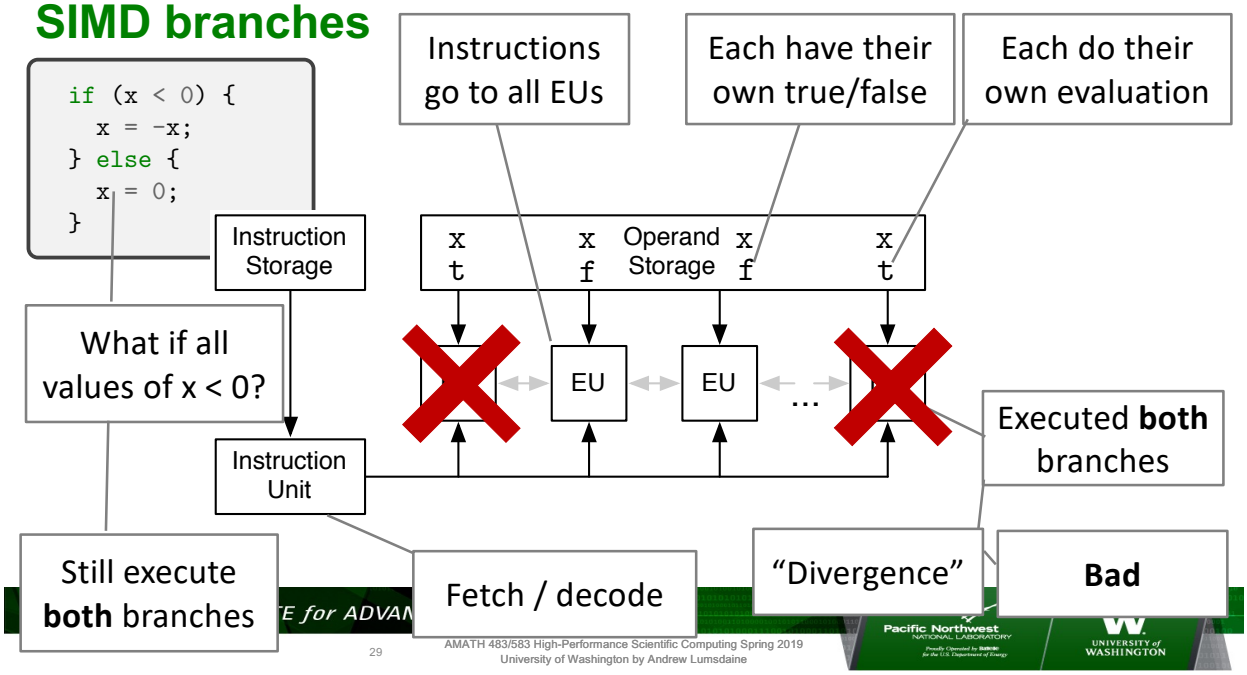



# SIMD branches

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

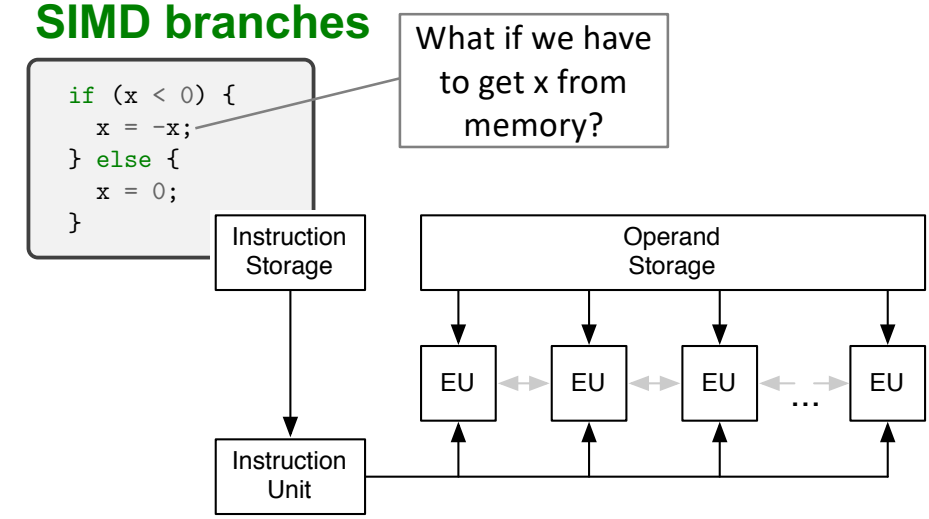


# SIMD branches

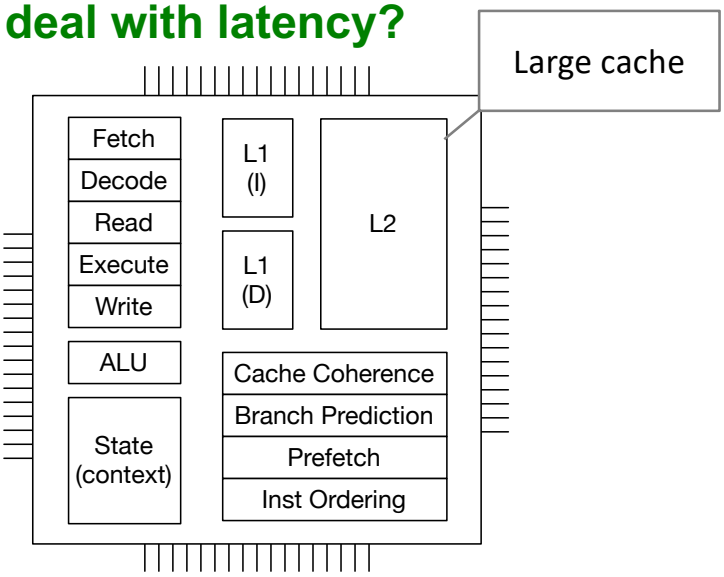





# SIMD branches

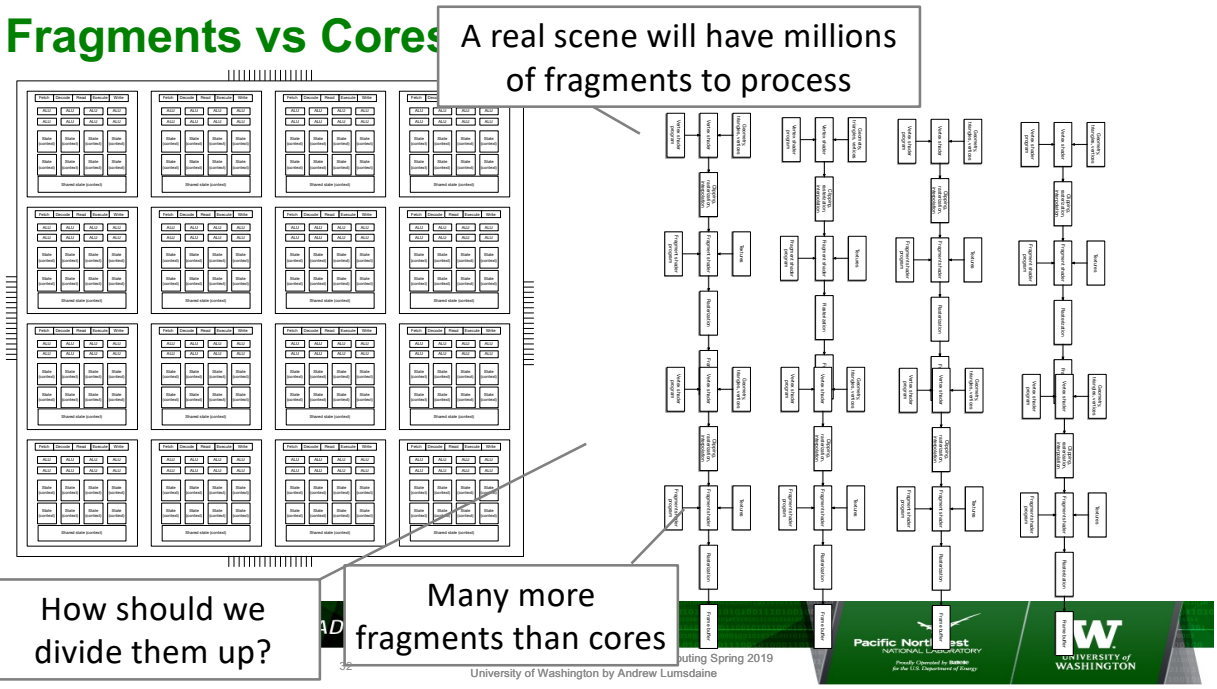


# How did CPUs deal with latency?

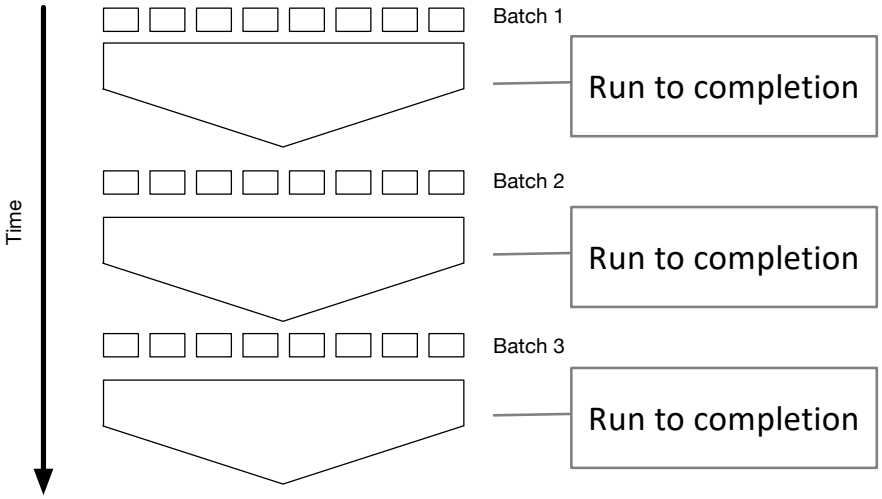





# Fragments vs Cores

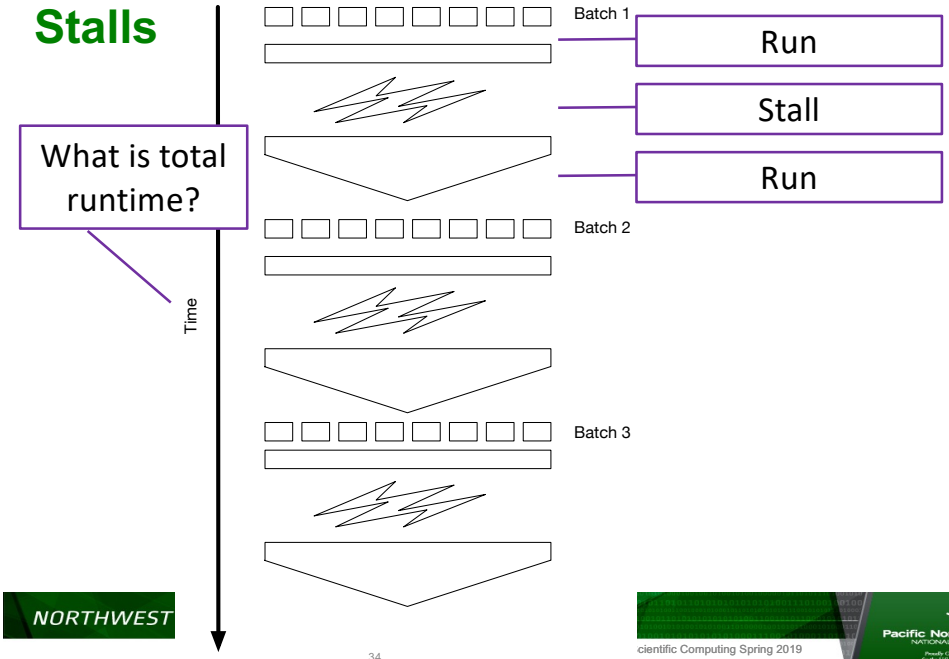


# Scheduling

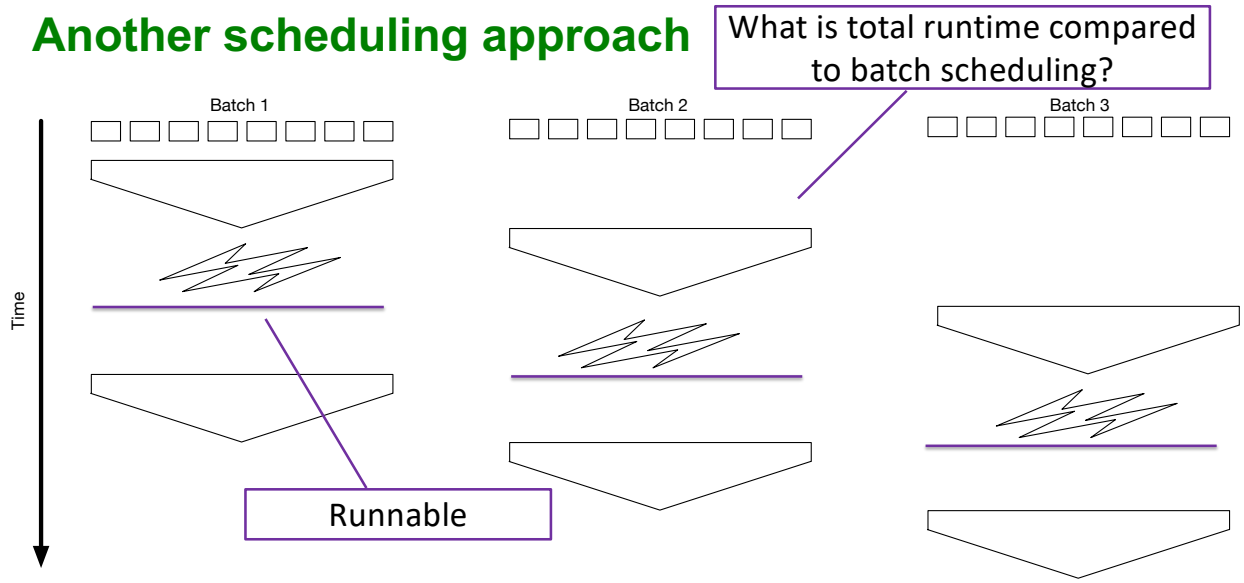





# Stalls



# Another scheduling approach






# GPU

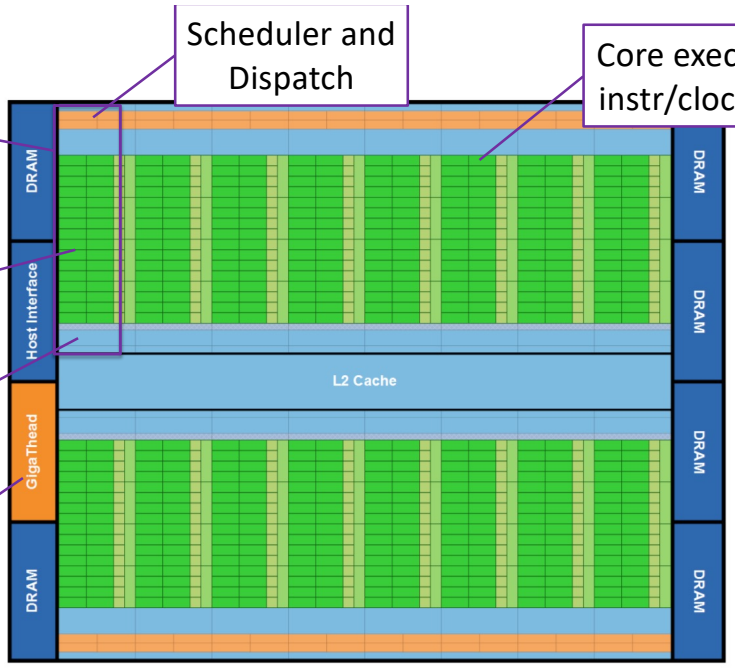
# Fermi

Streaming Multiprocessor (SM) (16)

32 cores per SM

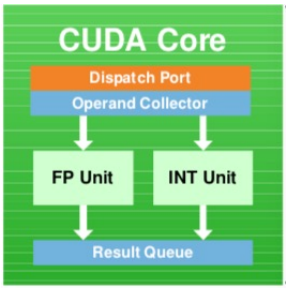
Register file and L1 cache

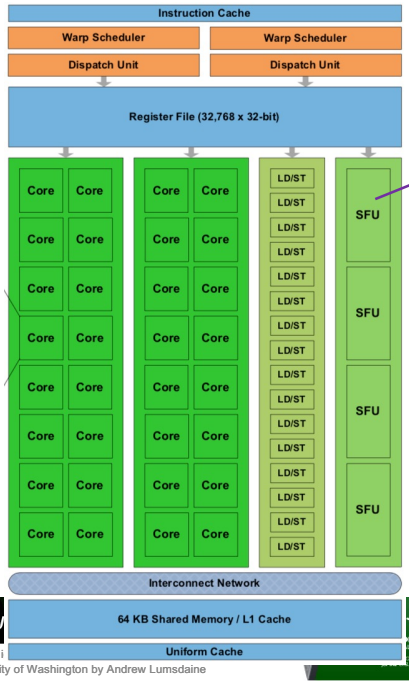
Global scheduler






# Fermi SM and Core



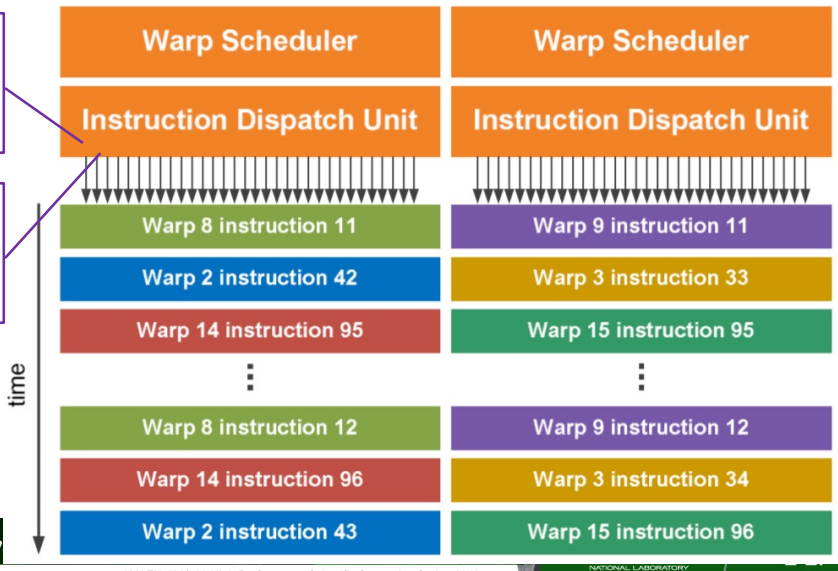

Special Function Unit

## Dual Warp Scheduler

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

One instruction from each warp

Up to 48* warps per SM






## For what are GPUs optimized?

- GPUs are optimized for _____

# Graphics Pipelines



# Fragment shader

```
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;
}
```

## Compute = Drawing, Kernel = Shader

```
                                float saxpy (
                                    float2 coords : TEXCOORD0,
                                    uniform sampler2D textureY,
                                    uniform sampler2D textureX,
                                    uniform float alpha ) : COLOR
                                {
                                    float result;
float yval=y_old[i];                float y = tex2D(textureY,coords);
float xval=x[i];                    float x = tex2D(textureX,coords);
y_new[i]=yval+alpha*xval;           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

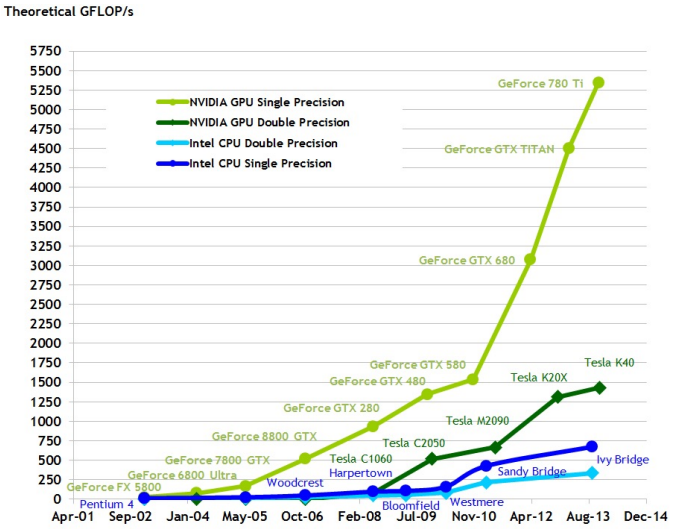
```
// set texture parameters
  glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,
                  GL_TEXTURE_MIN_FILTER, GL_NEAREST);
  glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,
                  GL_TEXTURE_MAG_FILTER, GL_NEAREST);
  glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,
                  GL_TEXTURE_WRAP_S, 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;
}
```



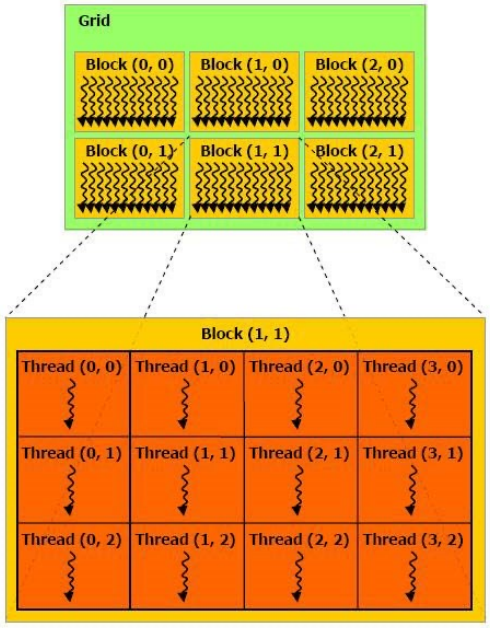
# Pain vs Gain




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

## 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



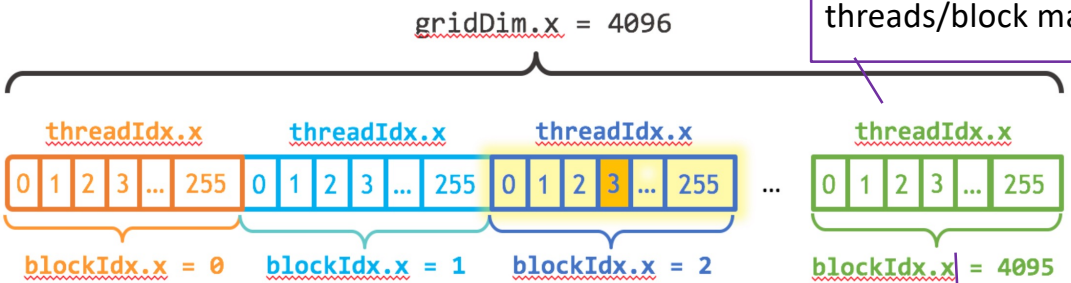

**Logical** organization



## Threads and blocks

At least 64

512 or 1024 threads/block max

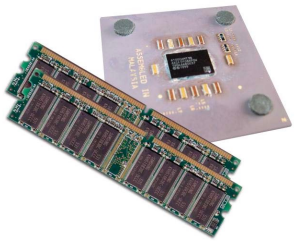



```
index = blockIdx.x  * blockDim.x + threadIdx.x
index =     (2)     *    (256)   +     (3)     = 515
```

Multiple of warp size

# GPU programming

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

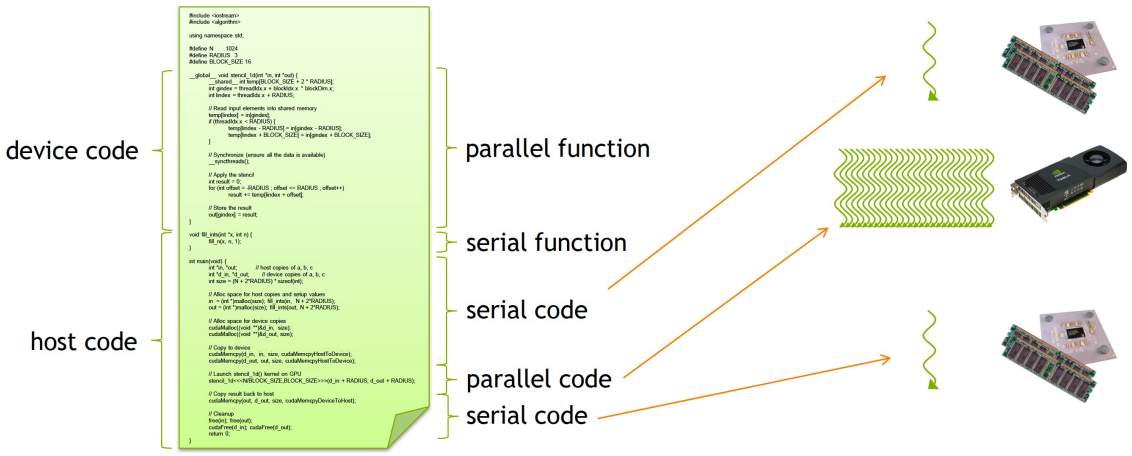



- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

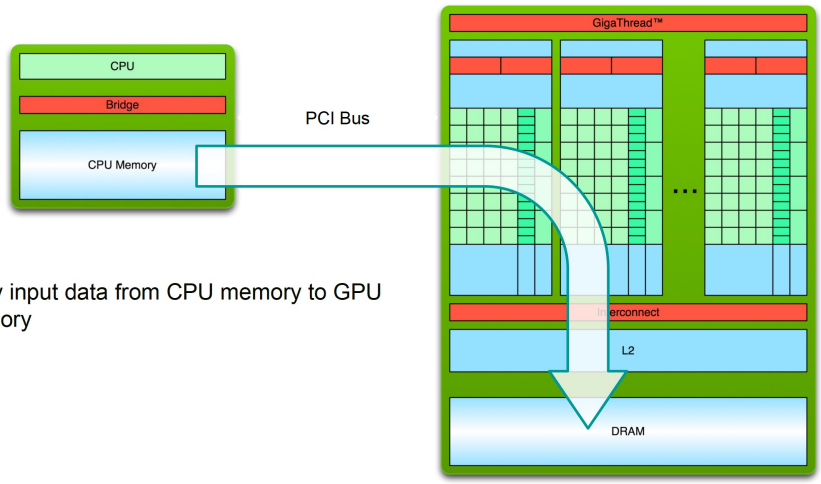




# Heterogeneous computing



# Simple processing flow

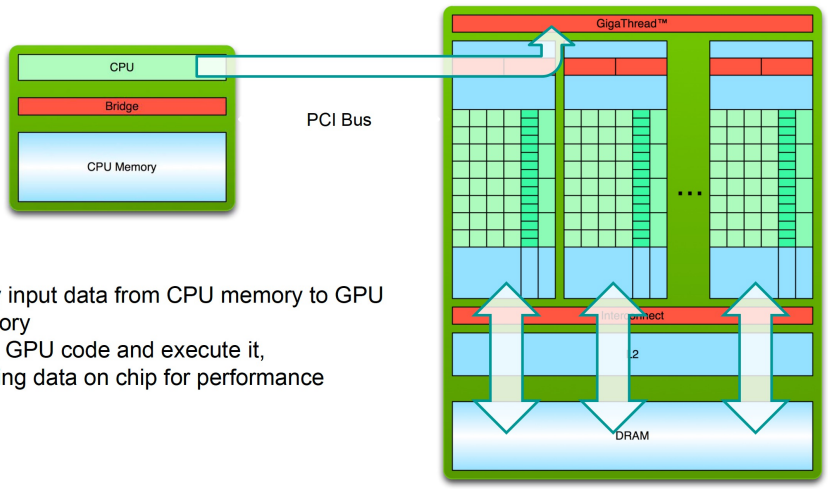



1. Copy input data from CPU memory to GPU memory



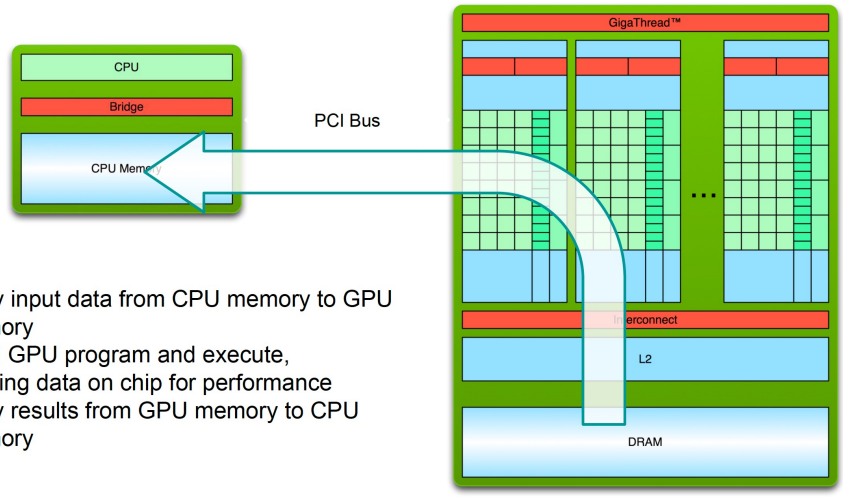


# Simple processing flow




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

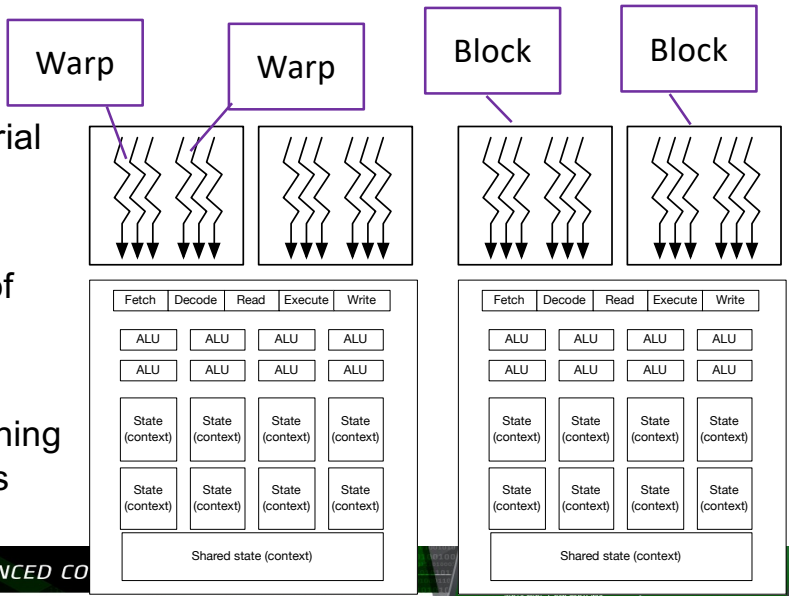




# Mapping software to hardware

- 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



## 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**

Will this work?

Execute on device

Execution configuration

Pointers! Bad!

Malloc! Bad!

Compile with nvcc

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

Do we want 256 threads doing this?

Recall partitioned two norm

# Blocks

# Threads

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

Partition based on thread id

```
__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

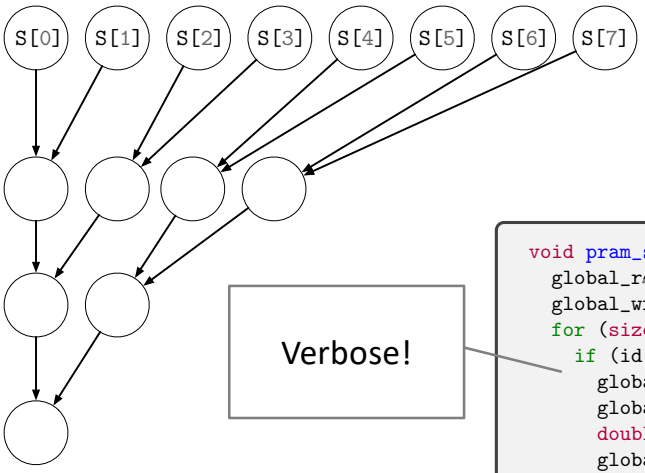

| Version | Laptop (GeForce GT 750M) | | Server (Tesla K80) | |
|---|---|---|---|---|
| | Time | Bandwidth | Time | Bandwidth |
| 1 CUDA Thread | 411ms | 30.6 MB/s | 463ms | 27.2 MB/s |
| 1 CUDA Block | 3.2ms | 3.9 GB/s | 2.7ms | 4.7 GB/s |
| Many CUDA Blocks | 0.68ms | 18.5 GB/s | 0.094ms | 134 GB/s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




# PRAM Sum

```
void pram_sum () {
  global_read (A[i], a);
  global_write(a, B[i]);
  for (size_t h = 0; h < log2(N); ++h) {
    if (id < (N / (2 << h))) {
      global_read(B[2i  ], x);
      global_read(B[2i+1], y);
      double z = x + y;
      global_write(z, B[i]);
    }
  }
}
```

# PRAM Sum



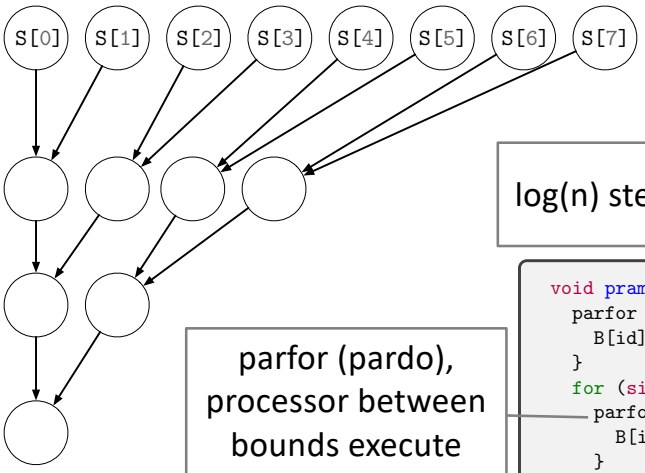

Simplify read/write with assignment

Verbose!

```
void pram_sum () {
  global_read (A[i], a);
  global_write(a, B[i]);
  for (size_t h = 0; h < log2(N); ++h) {
    if (id < (N / (2 << h))) {
      global_read(B[2i  ], x);
      global_read(B[2i+1], y);
      double z = x + y;
      global_write(z, B[i]);
    }
  }
}
```



# 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(blockI
    sdata[tid] = sdata[2*tid] + sdat
  }
}
```

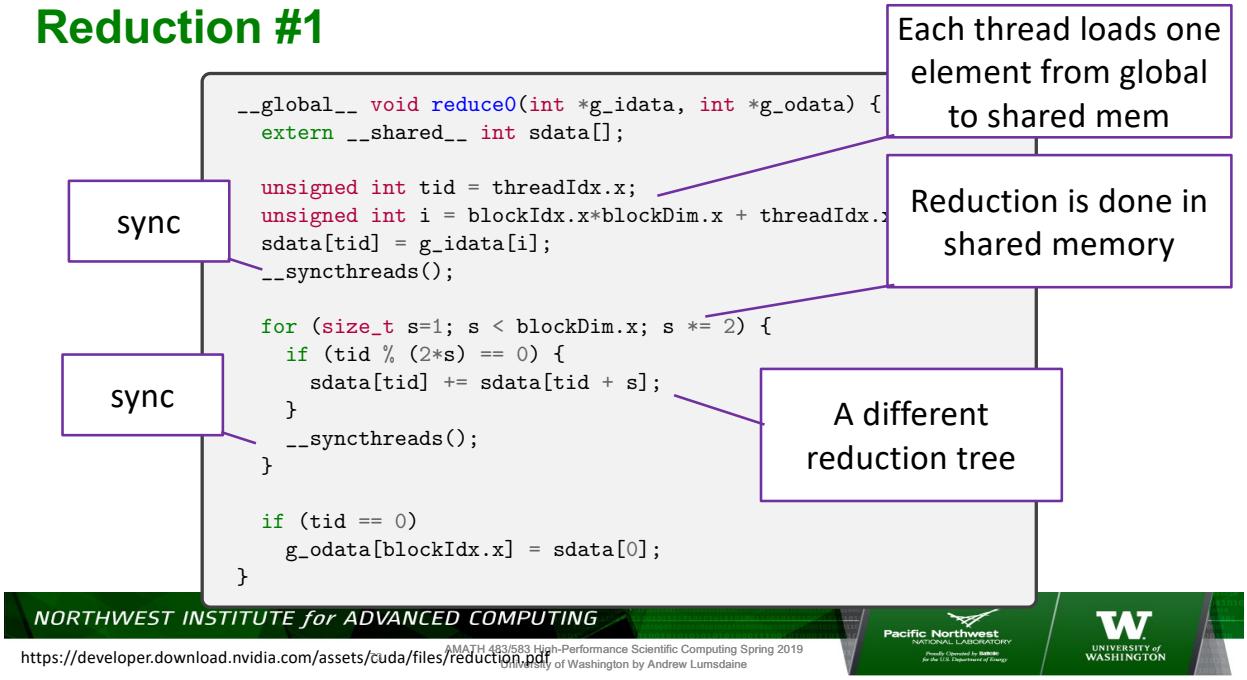
Copy answer back out

```
__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];
}
```

I thought threads were synchronized?

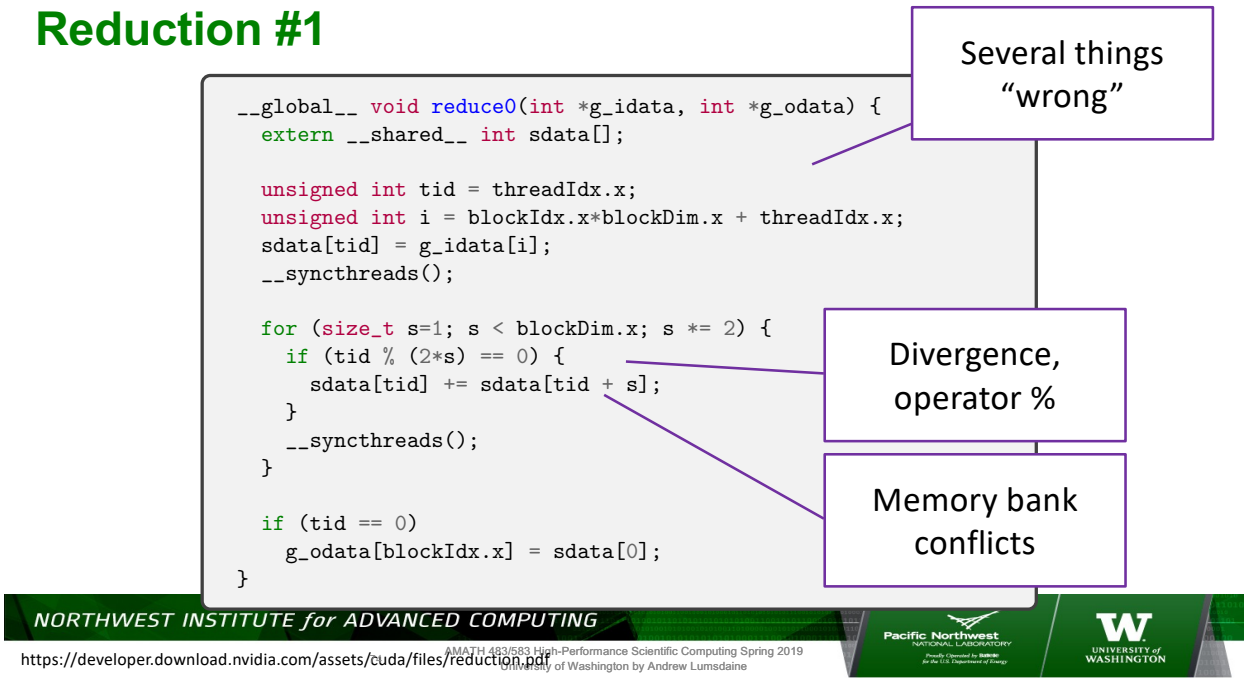
**Danger!**

# Reduction #1






# Reduction #1



# 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=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 |

# 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

Declare host_vector H with 4 elements

Initialize elements

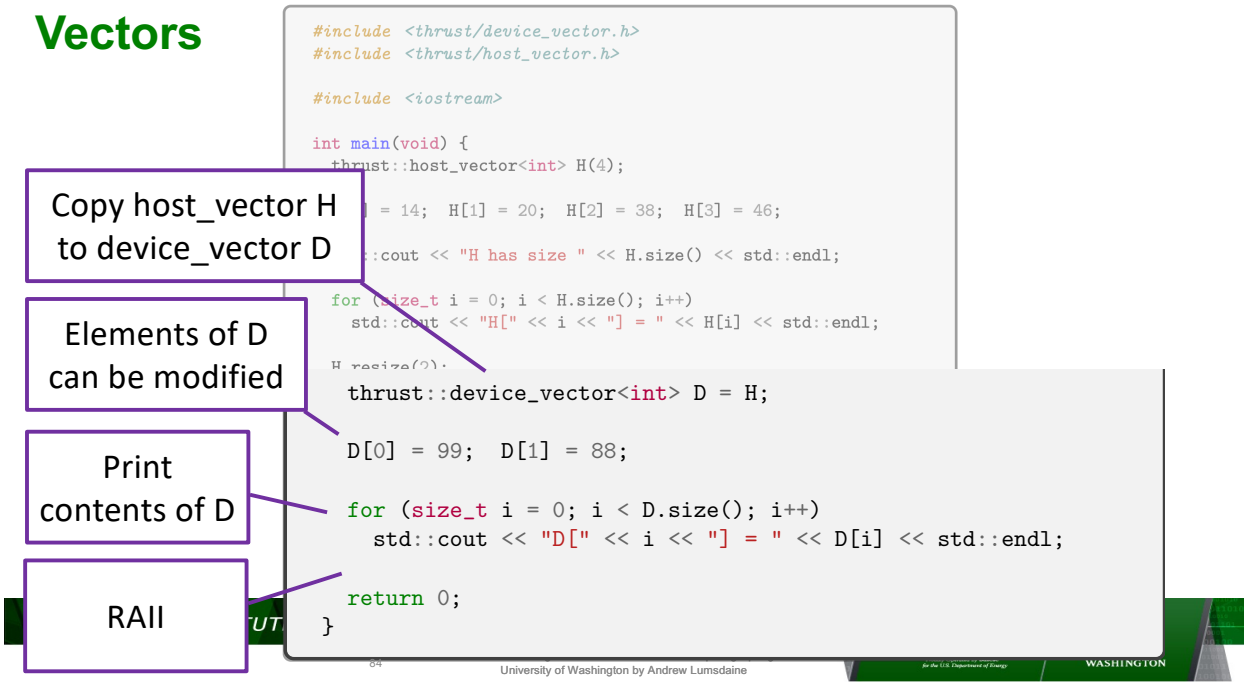
Print contents of H

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

#include <iostream>

int main(void) {
```

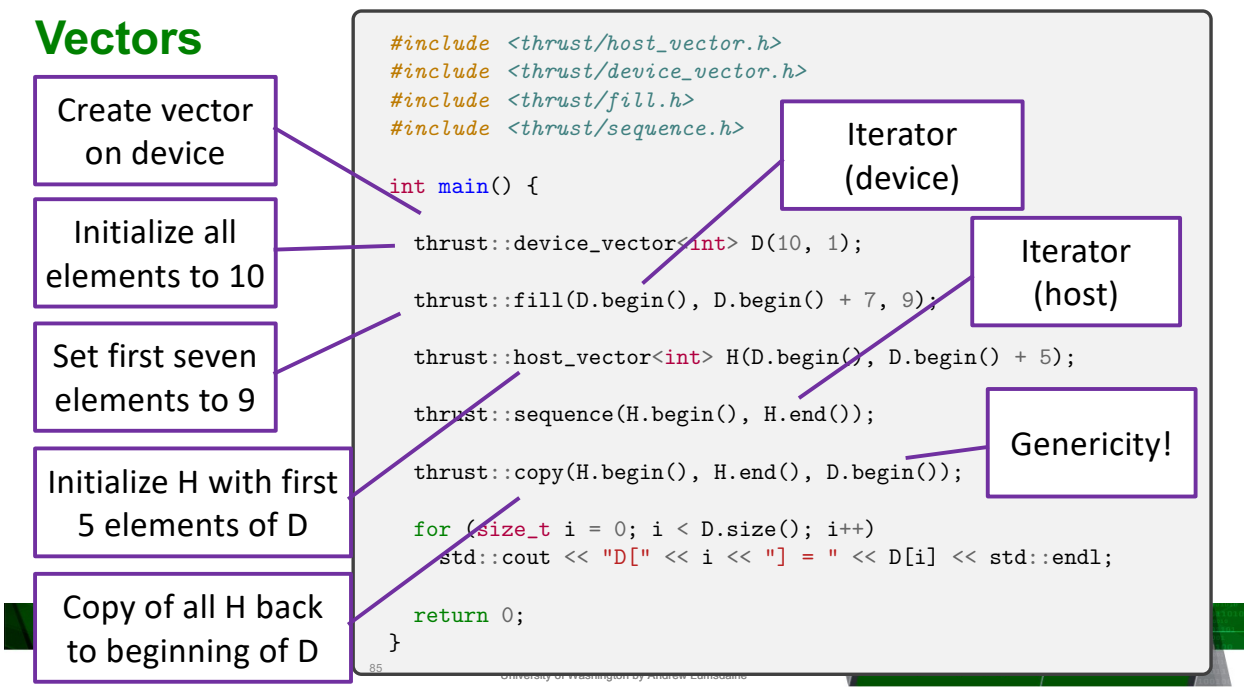
```
int main(void) {
  thrust::host_vector<int> H(4);

  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




## Vectors



# 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_x.begin(),
  ↪  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

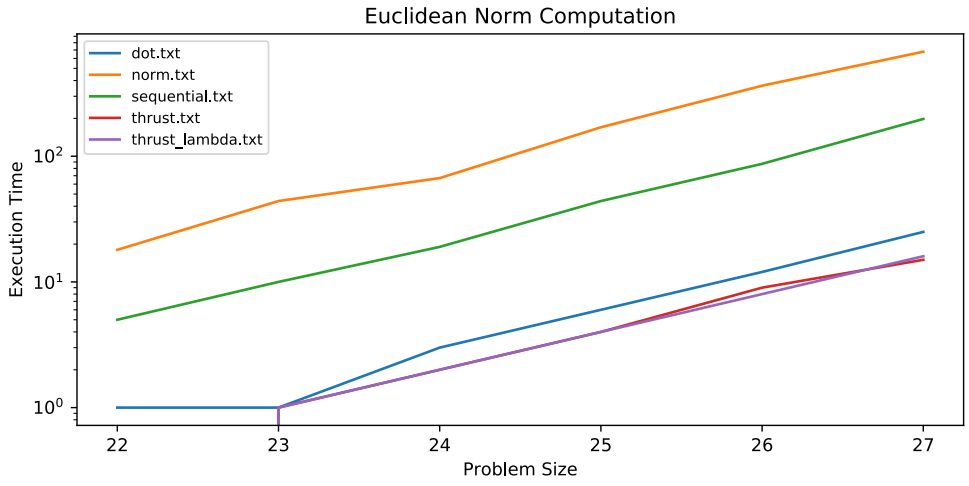
zip_iterator

# Execution policies

- Thrust allows GPU, CPU, OpenMP execution policies



# Performance



## Other GPU programming systems

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



## Thank you!