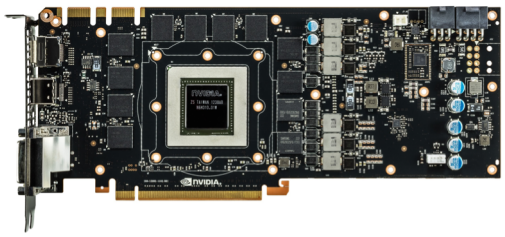
NORTHWEST INSTITUTE for ADVANCED COMPUTING

GPUs, CUDA, Python, Numba

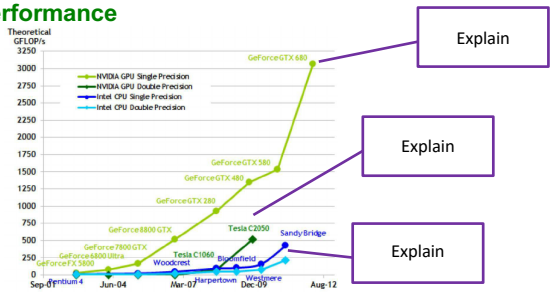
Andrew Lumsdaine
Northwest Institute for Advanced Computing
Pacific Northwest National Laboratory
University of Washington
Seattle, WA

GPU



NORTHWEST INSTITUTE for ADVANCED COMPUTING

Performance



Explain

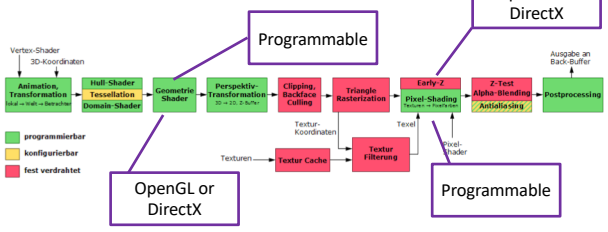
Explain

Explain

NORTHWEST INSTITUTE for ADVANCED COMPUTING

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.

GPU rendering pipeline



NORTHWEST INSTITUTE for ADVANCED COMPUTING

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();
```

NORTHWEST INSTITUTE for ADVANCED COMPUTING

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

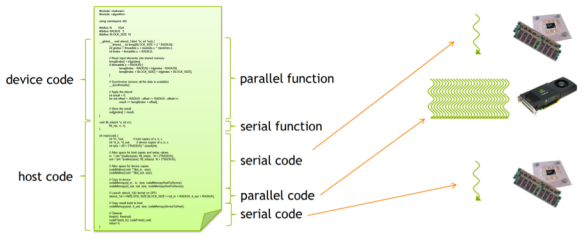
```
// transfer data to texture
glTexSubImage2D(GL_TEXTURE_RECTANGLE_ARB,0,0,0,texSize,texSize,
                GL_RGBA,GL_FLOAT,data);
// and read back
glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
glReadPixels(0, 0, texSize, texSize,GL_RGBA,GL_FLOAT,result);
// print out results
printf("Data before roundtrip:\n");
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;
}
```


GPU programming

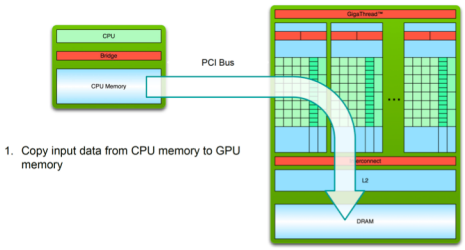
- Host: CPU and its memory
  - Host memory
- Device: GPU and its memory
  - Device memory

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

Heterogeneous computing

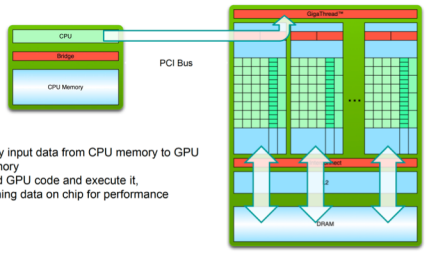


Simple processing flow



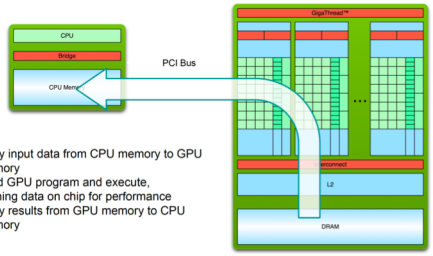
1. Copy input data from CPU memory to GPU memory

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

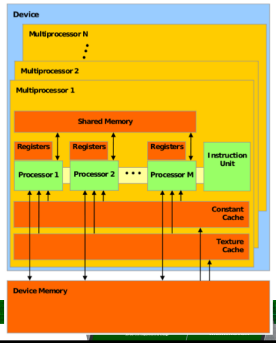


1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

GPU

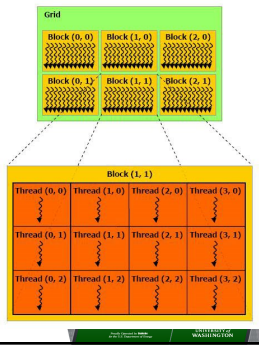
GPU architecture

- What are the important considerations re: performance?



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

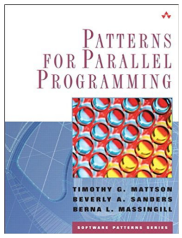


GPU Programming

- Transfer data from CPU memory to GPU memory
- Compute in GPU
- Transfer data back

- Computation is done with huge number of identical processing units
- Each PU executes exactly same kernel
- Partition the data among the processors – use thread idx, block idx, and block dimension to compute a unique (global)
- Use global index in the kernel to determine data each thread should work on

Parallelization Strategy



Timothy Mattson, Beverly Sanders, and Berna Massingill. 2004. *Patterns for Parallel Programming*(First ed.). Addison-Wesley Professional.

- Finding Concurrency
- Algorithm Structure
- Supporting Structures
- Implementation Mechanisms

NORTHWEST INSTITUTE for ADVANCED COMPUTING

Finding Concurrency

Decompose problem into pieces that can execute concurrently

Into tasks that can execute concurrently

Finding Concurrency

- Decomposition
  - Task Decomposition
  - Data Decomposition
- Dependency Analysis
  - Group Tasks
  - Order Tasks
  - Data Sharing
- Design Evaluation

Units that can be operated on (relatively) independently

- Algorithm Structure
- Supporting Structures
- Implementation Mechanisms

NORTHWEST INSTITUTE for ADVANCED COMPUTING

Finding Concurrency

Ways to group tasks to simplify management of dependencies

Finding Concurrency

- Decomposition
  - Task Decomposition
  - Data Decomposition
- Dependency Analysis
  - Group Tasks
  - Order Tasks
  - Data Sharing
- Design Evaluation

Ways to group tasks to simplify management of dependencies

Ways to order tasks for correctness, other constraints

Given a decomposition, ways to share data among tasks

- Algorithm Structure
- Supporting Structures
- Implementation Mechanisms

NORTHWEST INSTITUTE for ADVANCED COMPUTING

Algorithm Structure

Organize around concurrent tasks

Finding Concurrency

Fundamental organizing principle

Algorithm Structure

- Organize by Tasks
  - Task Parallelism
  - Divide and Conquer
- Organize by Data Decomposition
  - Geometric Decomposition
  - Recursive Data
- Organize by Flow of Data
  - Pipeline
  - Event-Based Coordination

Exploit potential concurrency in divide and conquer algorithms

- Supporting Structures
- Implementation Mechanisms

NORTHWEST INSTITUTE for ADVANCED COMPUTING

Algorithm Structure

Finding Concurrency

Fundamental organizing principle

Algorithm Structure

- Organize by Tasks
  - Task Parallelism
  - Divide and Conquer
- Organize by Data Decomposition
  - Geometric Decomposition
  - Recursive Data
- Organize by Flow of Data
  - Pipeline
  - Event-Based Coordination

Organize around a large data structure that is broken into "chunks"

Organize around operations on recursive data structure

- Supporting Structures
- Implementation Mechanisms

NORTHWEST INSTITUTE for ADVANCED COMPUTING

Algorithm Structure

Fundamental organizing principle

Finding Concurrency

Organize by sequence of independent stages

Algorithm Structure

- Organize by Tasks
  - Task Parallelism
  - Divide and Conquer
- Organize by Data Decomposition
  - Geometric Decomposition
  - Recursive Data
- Organize by Flow of Data
  - Pipeline
  - Event-Based Coordination

Organize by inherent communication among tasks

- Supporting Structures
- Implementation Mechanisms

NORTHWEST INSTITUTE for ADVANCED COMPUTING

Supporting Structures

Finding Concurrency

Algorithm Structure

Implementation Mechanisms

Organize communication and sharing between UEs

Centralized control distributing tasks

Translate loop bodies into tasks

Sets of dynamic tasks

Supporting Structures

Program Structures: SPMD, Manager/Worker, Loop Parallelism, Fork/Join

Data Structures: Shared Data, Shared Queue, Distributed Array

Explicitly manage shared data

Safely share a queue

Manage array data partitioned among UEs

Implementation Mechanisms

Finding Concurrency

Algorithm Structure

Supporting Structures

Implementation Mechanisms: UE Management, Synchronization, Communication

Manage task lifetimes

Enforce ordering constraints

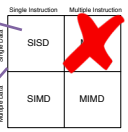
Get data where it needs to be when UEs don't share memory

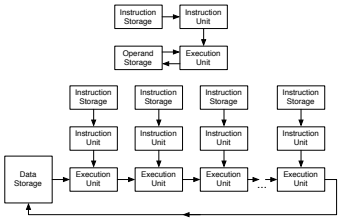
Flynn's Taxonomy (Aside)

Anyone in HPC must know Flynn's taxonomy

- **Classic** classification of parallel architectures (Michael Flynn, 1966)

Plain old sequential



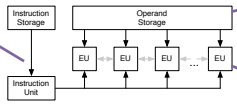


Based on multiplicity of instruction streams, data storage

SIMD and MIMD

- Two principal parallel computing paradigms (multiple

Single instruction at a time

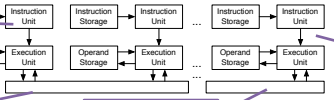


But each have their own data

Multiple instruction

All execution units execute in (c)lock step

EUs run independently (w own instrs)



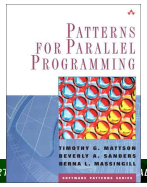
Coming up next

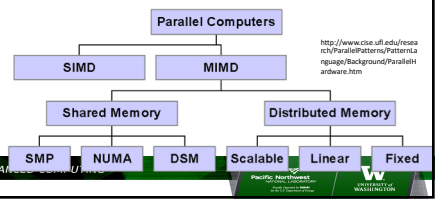
Shared Memory

Not Shared

A More Refined (Programmer-Oriented) Taxonomy

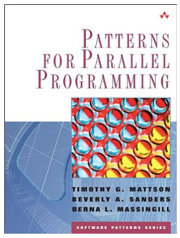
- Three major modes: SIMD, Shared Memory, Distributed Memory
- Different programming approaches are generally associated with different modes of parallelism (threads for shared, MPI for distributed)
- A modern supercomputer will have all three major modes present





http://www.cise.ufl.edu/research/ParallelPatterns/PatternLanguage/Background/ParallelHardware.htm

SIMD Parallelization Strategy



Timothy Mattson, Beverly Sanders, and Berna Massingill. 2004. *Patterns for Parallel Programming*(First ed.). Addison-Wesley Professional.

Finding Concurrency

Algorithm Structure

Supporting Structures

Implementation Mechanisms

This will often be loops

Refactor to eliminate obstacles to parallelism

Revise to use data parallel operations

Allocate elements to threads

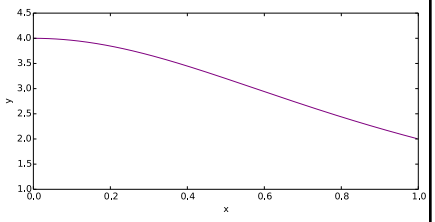
Write in CUDA

Tune

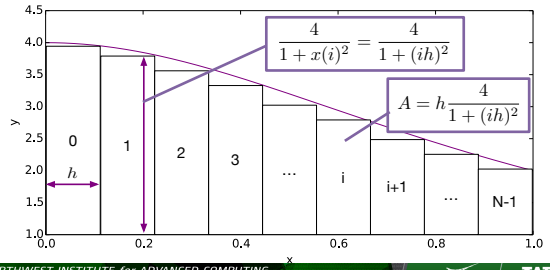
Example

- Find the value of π
- Using formula

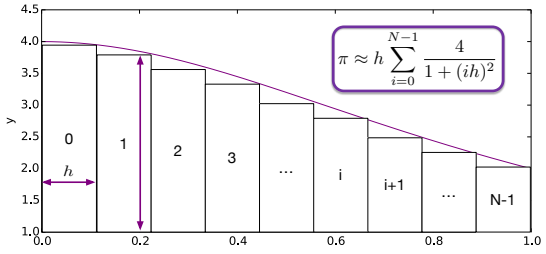
$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$



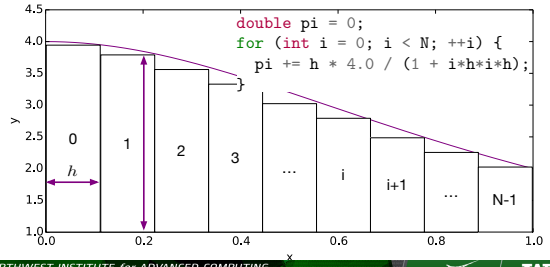
Numerical Quadrature



Numerical Quadrature

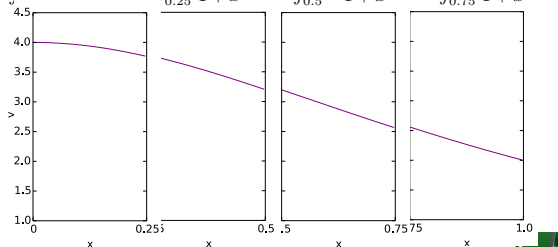


Numerical Quadrature (Sequential)



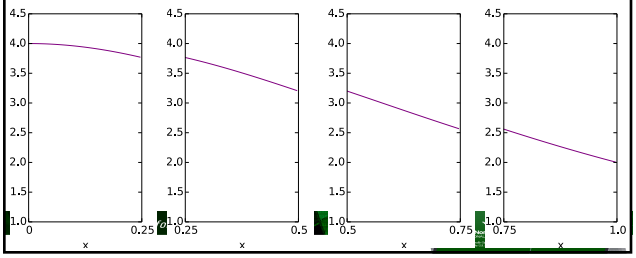
Finding Concurrency

$$\pi = \int_0^{0.25} \frac{4}{1+x^2} dx + \int_{0.25}^{0.5} \frac{4}{1+x^2} dx + \int_{0.5}^{0.75} \frac{4}{1+x^2} dx + \int_{0.75}^{1} \frac{4}{1+x^2} dx$$



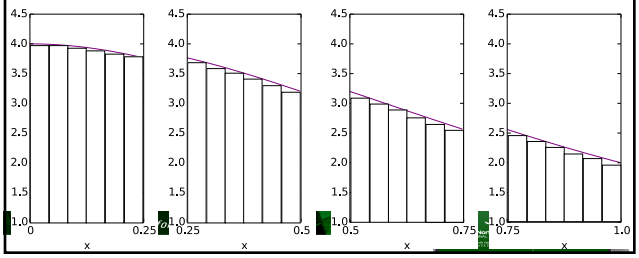
Finding Concurrency

$$\pi = \int_0^{0.25} \frac{4}{1+x^2} dx + \int_{0.25}^{0.5} \frac{4}{1+x^2} dx + \int_{0.5}^{0.75} \frac{4}{1+x^2} dx + \int_{0.75}^{1} \frac{4}{1+x^2} dx$$



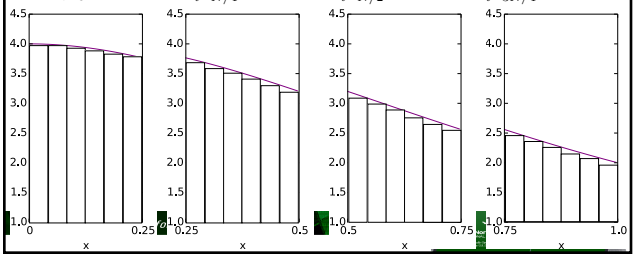
Finding Concurrency

$$\pi = \int_0^{0.25} \frac{4}{1+x^2}dx + \int_{0.25}^{0.5} \frac{4}{1+x^2}dx + \int_{0.5}^{0.75} \frac{4}{1+x^2}dx + \int_{0.75}^{1} \frac{4}{1+x^2}dx$$



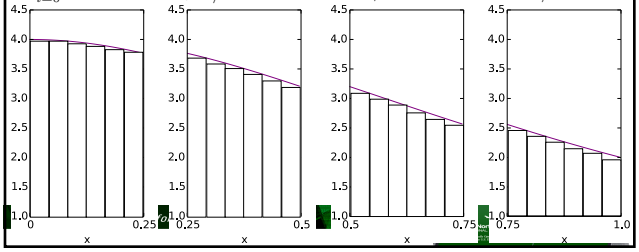
Finding Concurrency

$$\pi \approx h\sum_{i=0}^{N/4-1} \frac{4}{1+(ih)^2} + h\sum_{i=N/4}^{N/2-1} \frac{4}{1+(ih)^2} + h\sum_{i=N/2}^{3N/4-1} \frac{4}{1+(ih)^2} + h\sum_{i=3N/4}^{3N-1} \frac{4}{1+(ih)^2}$$



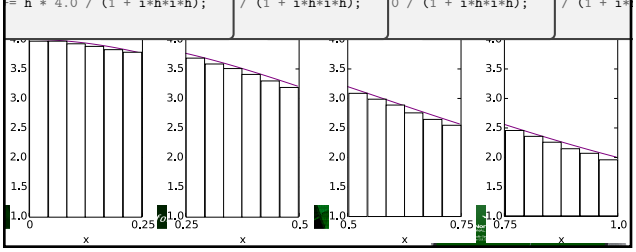
Finding Concurrency

$$h\sum_{i=0}^{N/4-1} \frac{4}{1+(ih)^2} \quad h\sum_{i=N/4}^{N/2-1} \frac{4}{1+(ih)^2} \quad h\sum_{i=N/2}^{3N/4-1} \frac{4}{1+(ih)^2} \quad h\sum_{i=3N/4}^{N-1} \frac{4}{1+(ih)^2}$$



Finding Concurrency

```
for (int i = begin; i < end; ++i) {
  pi += h * 4.0 / (1 + i*h*i*h);
}
```



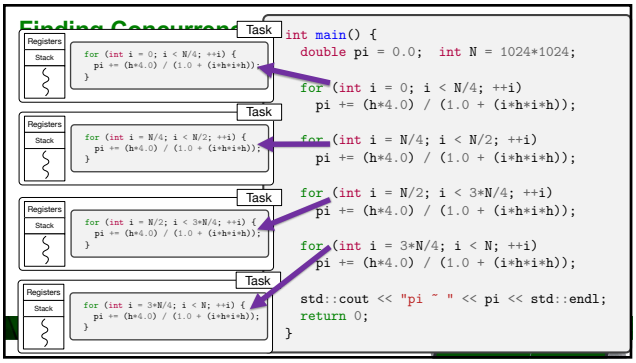
Finding Concurrency

$$h\sum_{i=0}^{N/4-1} \frac{4}{1+(ih)^2} \quad h\sum_{i=N/4}^{N/2-1} \frac{4}{1+(ih)^2} \quad h\sum_{i=N/2}^{3N/4-1} \frac{4}{1+(ih)^2} \quad h\sum_{i=3N/4}^{N-1} \frac{4}{1+(ih)^2}$$

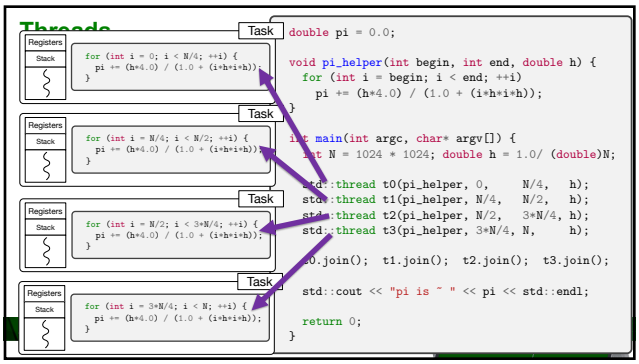
```
int main() {
  double pi = 0.0;  int N = 1024*1024;

  for (int i = 0; i < N/4; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = N/4; i < N/2; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = N/2; i < 3*N/4; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = 3*N/4; i < N; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  std::cout << "pi ~ " << pi << std::endl;
  return 0;
}
```

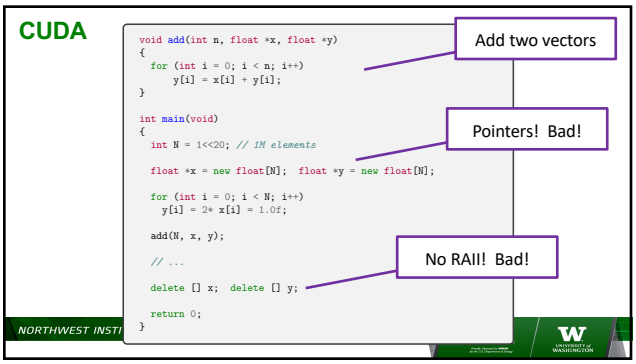
NORTHWEST INSTITUTE for ADVANCED COM

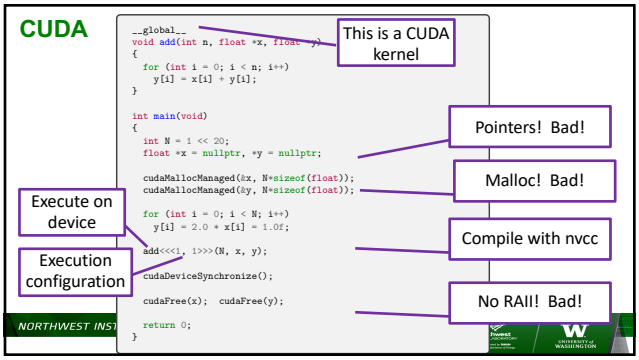
Finding Concurrency

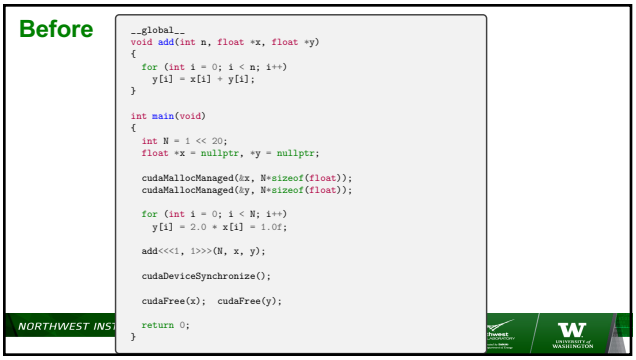


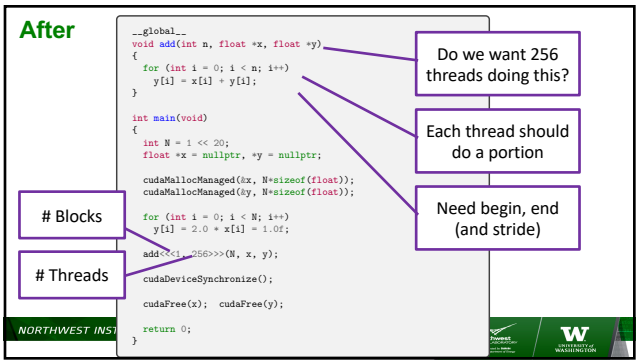

```
int main() {
  double pi = 0.0;  int N = 1024*1024;

  for (int i = 0; i < N/4; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = N/4; i < N/2; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = N/2; i < 3*N/4; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  for (int i = 3*N/4; i < N; ++i)
    pi += (h*4.0) / (1.0 + (i*h*i*h));

  std::cout << "pi ~ " << pi << std::endl;
  return 0;
}
```

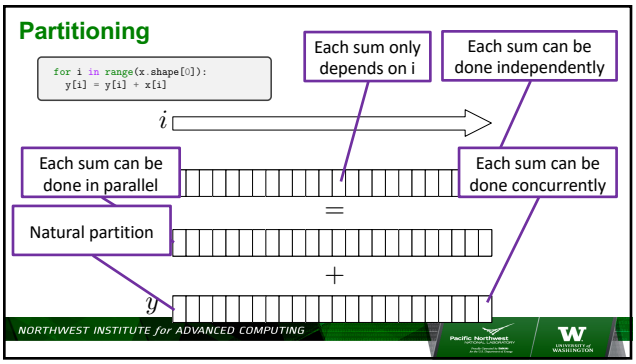

Threads
Task
Registers
Stack
for (int i = 0; i < N/4; ++i) {
pi += (h*4.0) / (1.0 + (i+h*i+h));
}
Task
Registers
Stack
for (int i = N/4; i < N/2; ++i) {
pi += (h*4.0) / (1.0 + (i+h*i+h));
}
Task
Registers
Stack
for (int i = N/2; i < 3*N/4; ++i) {
pi += (h*4.0) / (1.0 + (i+h*i+h));
}
Task
Registers
Stack
for (int i = 3*N/4; i < N; ++i) {
pi += (h*4.0) / (1.0 + (i+h*i+h));
}
double pi = 0.0;
void pi_helper(int begin, int end, double h) {
for (int i = begin; i < end; ++i)
pi += (h*4.0) / (1.0 + (i*h*i*h));
}
int main(int argc, char* argv[]) {
int N = 1024 * 1024; double h = 1.0/ (double)N;
std::thread t0(pi_helper, 0, N/4, h);
std::thread t1(pi_helper, N/4, N/2, h);
std::thread t2(pi_helper, N/2, 3*N/4, h);
std::thread t3(pi_helper, 3*N/4, N, h);
t0.join(); t1.join(); t2.join(); t3.join();
std::cout << "pi is " << pi << std::endl;
return 0;
}


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;
}
Add two vectors
Pointers! Bad!
No RAII! Bad!
NORTHWEST INSTI

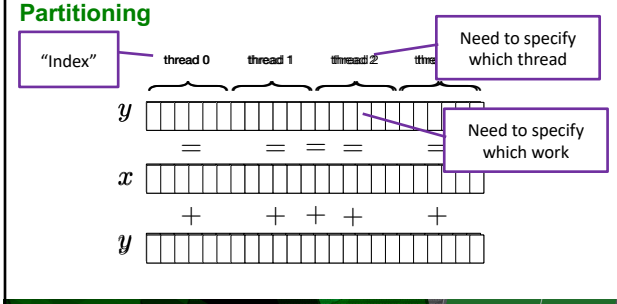

CUDA
__global__
void add(int n, float *x, float *y)
{
for (int i = 0; i < n; i++)
y[i] = x[i] + y[i];
}
int main(void)
{
int N = 1 << 20;
float *x = nullptr, *y = nullptr;
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));
for (int i = 0; i < N; i++)
y[i] = 2.0 * x[i] = 1.0f;
add<<<1, 1>>>(N, x, y);
cudaDeviceSynchronize();
cudaFree(x); cudaFree(y);
return 0;
}
This is a CUDA kernel
Pointers! Bad!
Malloc! Bad!
Compile with nvcc
No RAII! Bad!
Execute on device
Execution configuration
NORTHWEST INST


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;
}
NORTHWEST INST


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;
}
Do we want 256 threads doing this?
Each thread should do a portion
Need begin, end (and stride)
Blocks
Threads
NORTHWEST INST

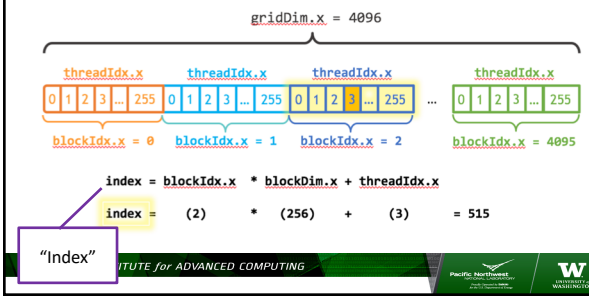

Partitioning
for i in range(x.shape[0]):
y[i] = y[i] + x[i]
Each sum only depends on i
Each sum can be done independently
i
Each sum can be done in parallel
Each sum can be done concurrently
=
Natural partition
+
y
NORTHWEST INSTITUTE for ADVANCED COMPUTING

Partitioning



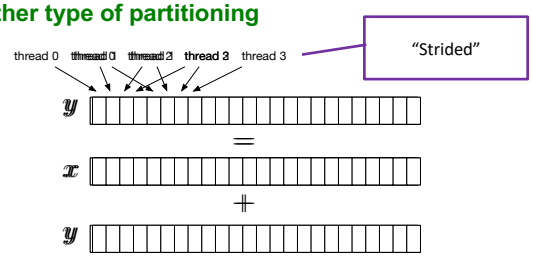
NORTHWEST INSTITUTE for ADVANCED COMPUTING

Threads and blocks



NORTHWEST INSTITUTE for ADVANCED COMPUTING

Another type of partitioning



NORTHWEST INSTITUTE for ADVANCED COMPUTING

Partitioned

```
__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

NORTHWEST INSTITUTE for ADVANCED COMPUTING

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;
}
```

NORTHWEST INSTITUTE for ADVANCE

Blocks

Index taking into account blocks

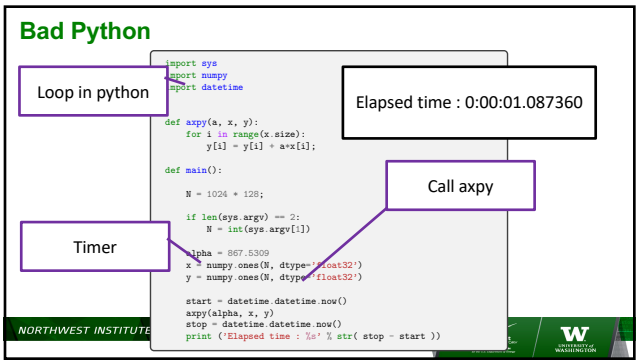
Stride taking into account blocks

Multiple blocks

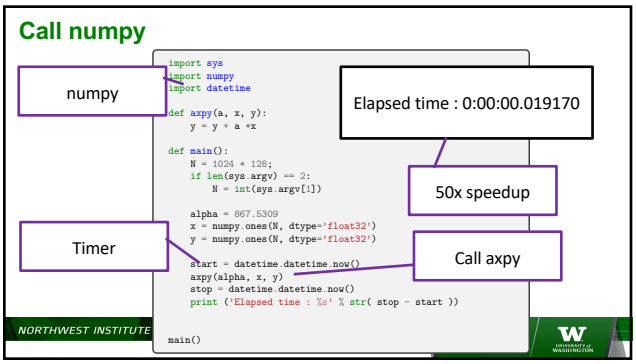
```
__global__
void add(int n, float *x, float *y){
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[I];
}

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;
}
```

NORTHWEST INSTITUTE for ADVANCED CO

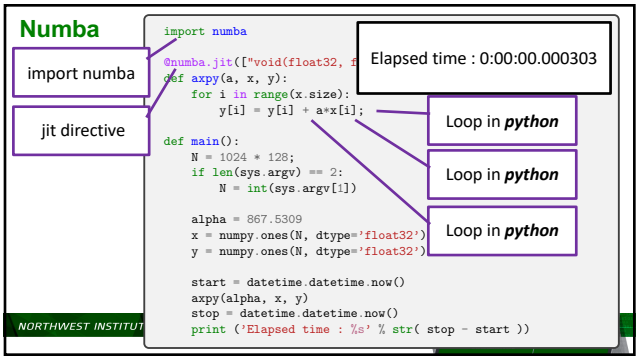
Bad Python



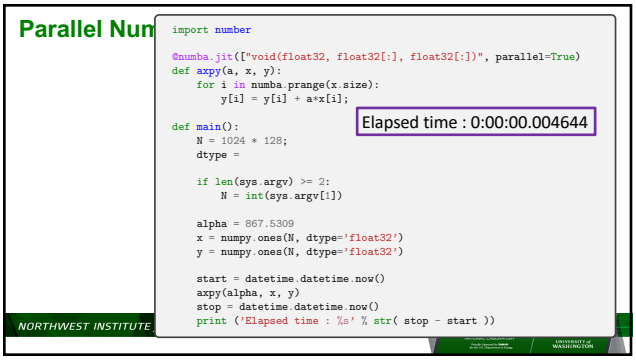
Call numpy



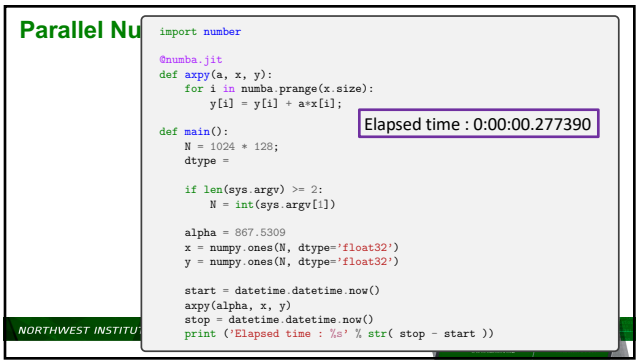
Numba



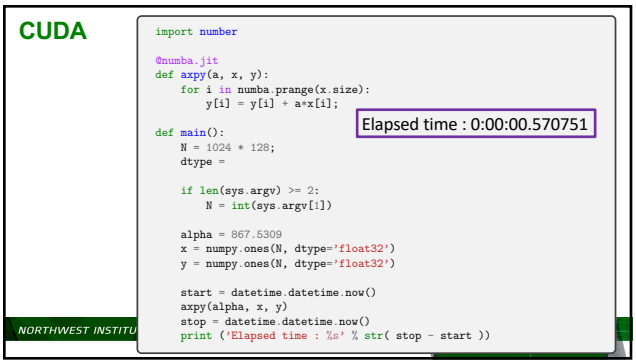
Parallel Num



Parallel Nu



CUDA



CUDA

```
import numba
from numba import cuda

@cuda.jit
def axpy(a, x, y):
    tx = cuda.threadIdx.x
    ty = cuda.blockIdx.x
    bw = cuda.blockDim.x
    pos = tx + ty * bw

    if pos < x.size:
        y[pos] = y[pos] + a*x[pos]

def main():
    N = 1024 * 128;

    if len(sys.argv) >= 2:
        N = int(sys.argv[1])

    alpha = 1.000000001;
    x = numpy.ones(N, dtype=dtype)
    y = numpy.ones(N, dtype=dtype)

    y[0] = 0;  y[N-1] = 0;
    start = datetime.datetime.now()
    axpy[(N+255)//256, 256](alpha, x, y)
    stop = datetime.datetime.now()
    print ('Elapsed time : %s' % (str( stop - start )))
```

Elapsed time : 0:00:00.21673

Performance

V.1 V.2

Version	Laptop (GeForce GT 750M) Time	Bandwidth	Server (Tesla K80) 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

V.3 5000X !!

Summary (so far)

- Transfer data from CPU memory to GPU memory
- Compute in GPU
- Transfer data back

- Computation is done with huge number of identical processing units
- Each PU executes exactly same kernel
- Partition the data among the processors – use thread idx, block idx, and block dimension to compute a unique (global)
- Use global index in the kernel to determine data each thread should work on

Other GPU programming systems

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

Thank you!

Creative Commons BY-NC-SA 4.0 License