Introduction to CUDA C

A Tutorial

Jayant Apte, ASPITRG

Introduction to CUDA C
A Tutorial

Jayant Apte, ASPITRG

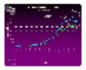
Prezi

	Single Data	Multiple Data
Single Instruction	SISD typical thread	SIMD vector processors GPUs SSE instructions
Multiple Instruction	MISD rare possibly parts of [illegible]	MIMD cluster of computers

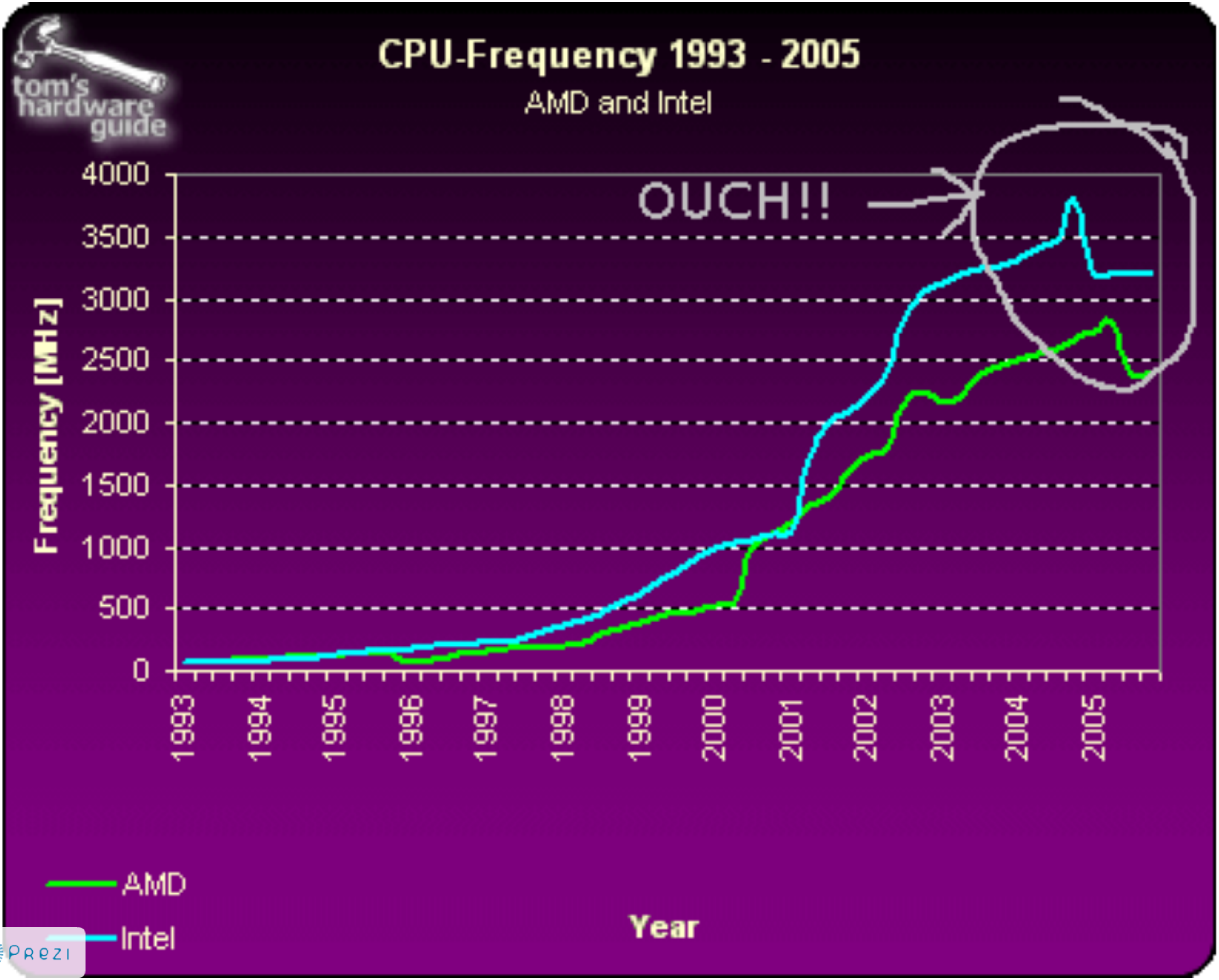
Parallel Computing

Why do we care?

Why build parallel systems?



How about we automatically convert serial programs to parallel programs?



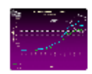
tom's hardware guide
CPU-Frequency 1993 - 2005
AMD and Intel
OUCH!!
Frequency [MHz]
4000
3500
3000
2500
2000
1500
1000
500
0
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
Year
AMD
Intel

	Single Data	Multiple Data
Single Instruction	SISD typical thread	SIMD vector processors GPUs SSE instructions
Multiple Instruction	MISD rare possibly sets of filters	MIMD cluster of computers

Parallel Computing

Why do we care?

Why build parallel systems?



How about we automatically convert serial programs to parallel programs?

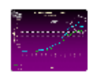
- *Very limited success with converting C/C++ code to parallel code.*
- *Step back. Rework the algorithm.*

	Single Data	Multiple Data
Single Instruction	SISD typical thread	SIMD vector processors GPUs SSE instructions
Multiple Instruction	MISD rare possibly set of [illegible]	MIMD cluster of computers

Parallel Computing

Why do we care?

Why build parallel systems?

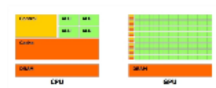


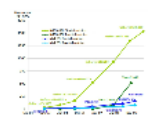
How about we automatically convert serial programs to parallel programs?

	Single Data	Multiple Data
Single Instruction	*SISD* typical thread	*SIMD* vector processors GPUs SSE instructions
Multiple Instruction	*MISD* rare possibly set of filters	*MIMD* cluster of computers

Comparison with CPU

- Cache sizes
- Floating point capability

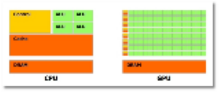


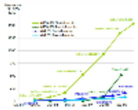


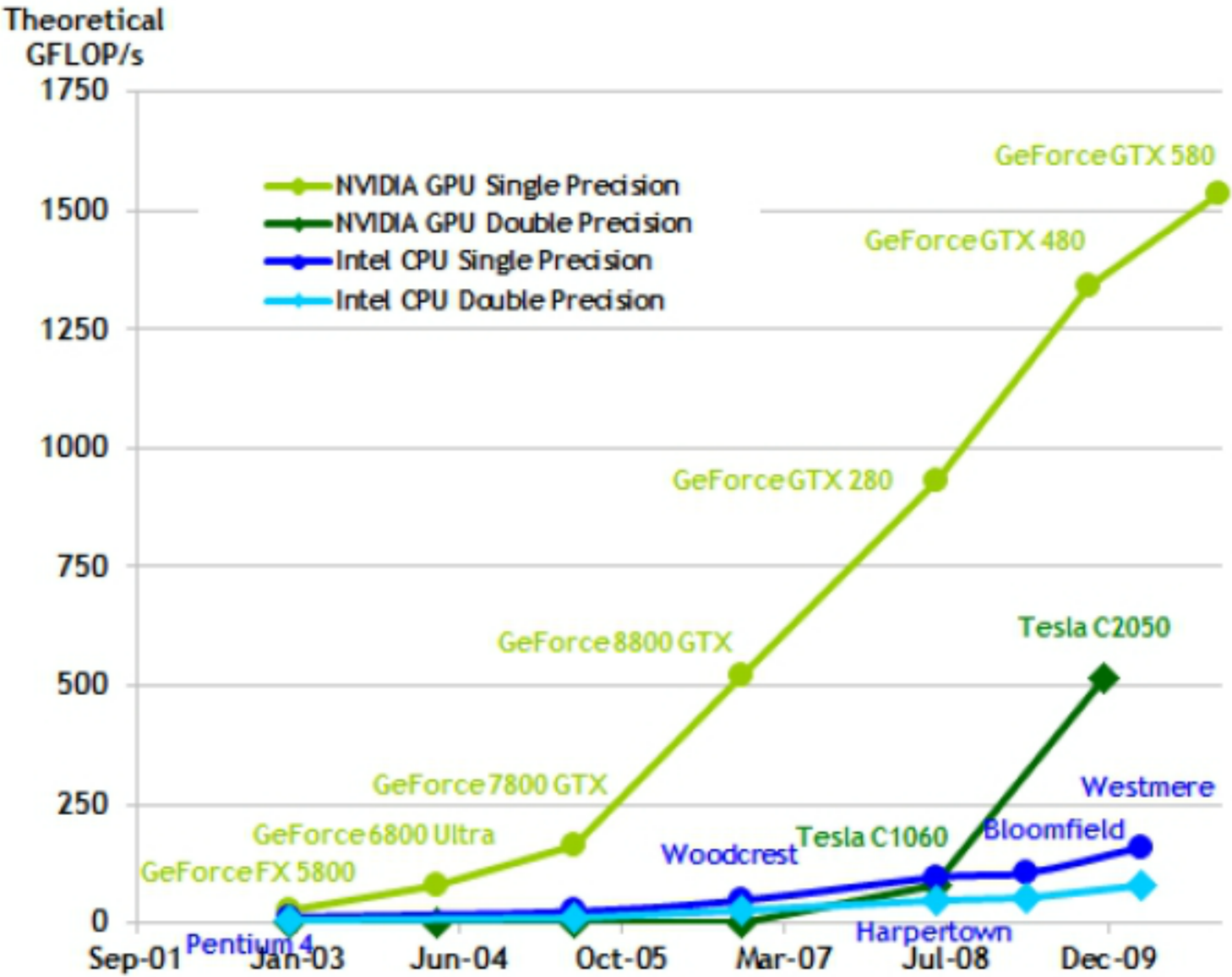

Control
ALU
ALU
ALU
ALU
Cache
DRAM
CPU
DRAM
GPU

Comparison with CPU

- Cache sizes
- Floating point capability

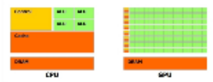


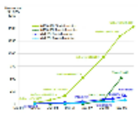



Theoretical GFLOP/s
1750
1500
1250
1000
750
500
250
0
NVIDIA GPU Single Precision
NVIDIA GPU Double Precision
Intel CPU Single Precision
Intel CPU Double Precision
GeForce GTX 580
GeForce GTX 480
GeForce GTX 280
GeForce 8800 GTX
GeForce 7800 GTX
GeForce 6800 Ultra
GeForce FX 5800
Tesla C2050
Tesla C1060
Westmere
Bloomfield
Woodcrest
Harpertown
Pentium 4
Sep-01
Jan-03
Jun-04
Oct-05
Mar-07
Jul-08
Dec-09
Prezi

Comparison with CPU

- Cache sizes
- Floating point capability

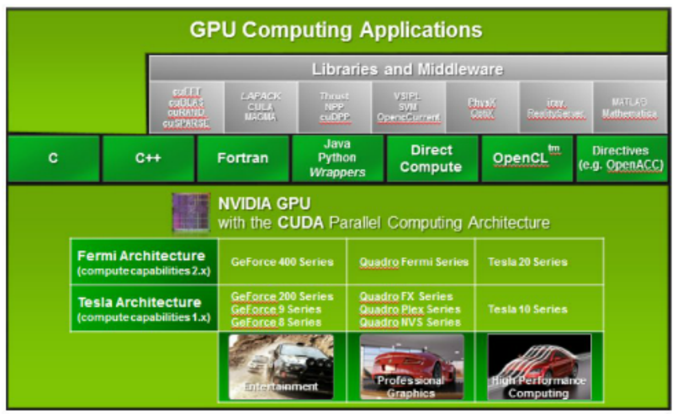




Prezi

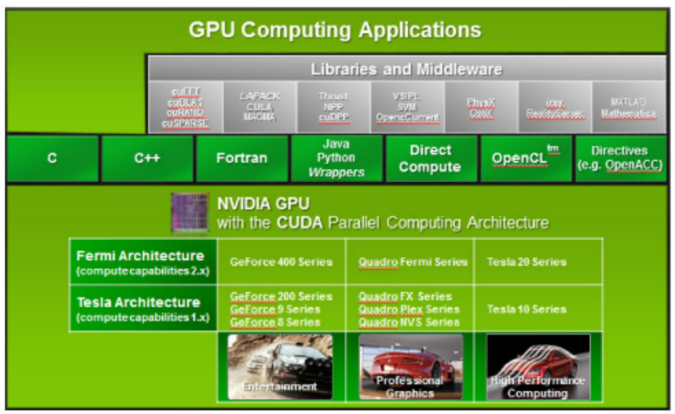
CUDA

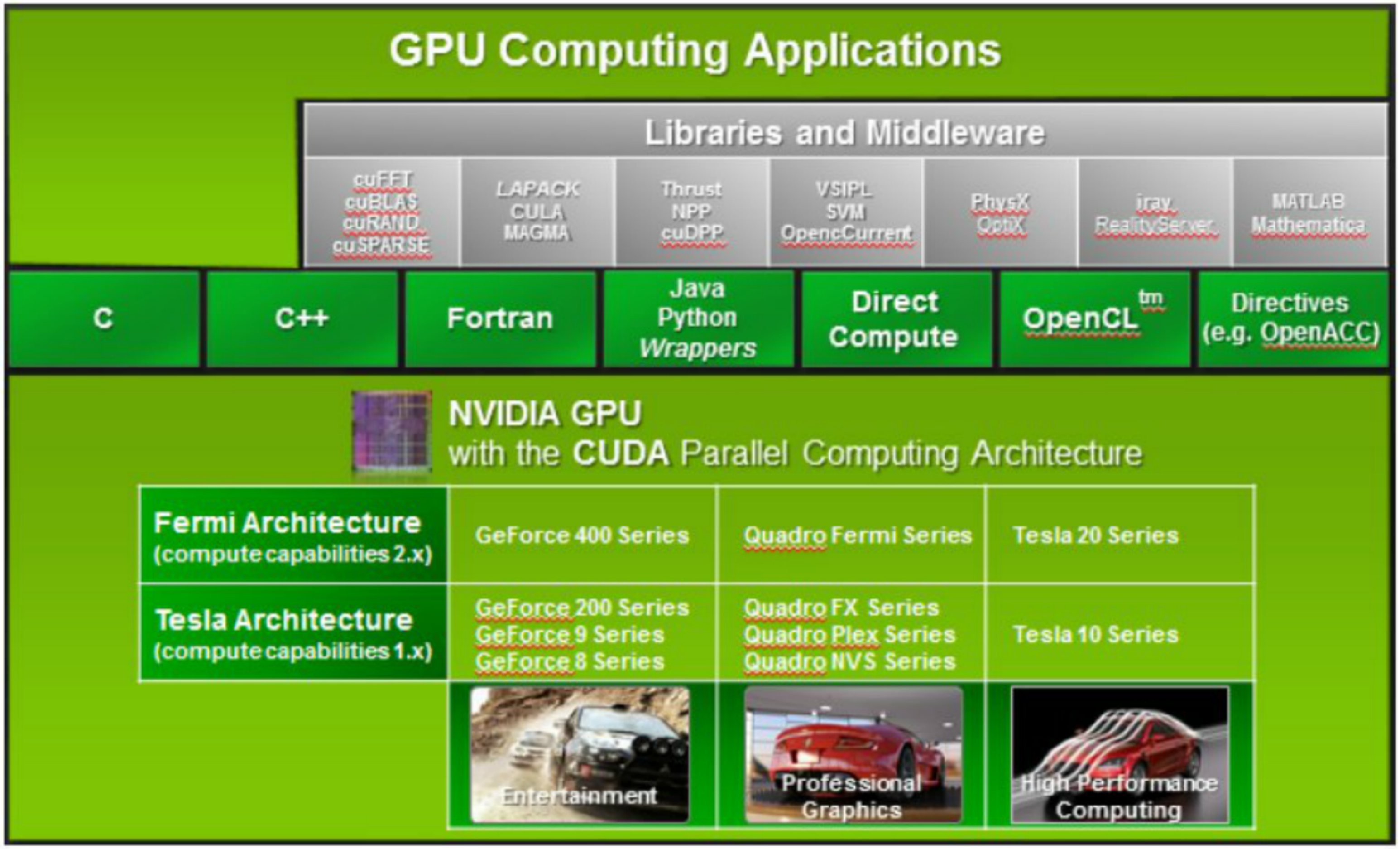
Compute Unified Device Architecture



CUDA

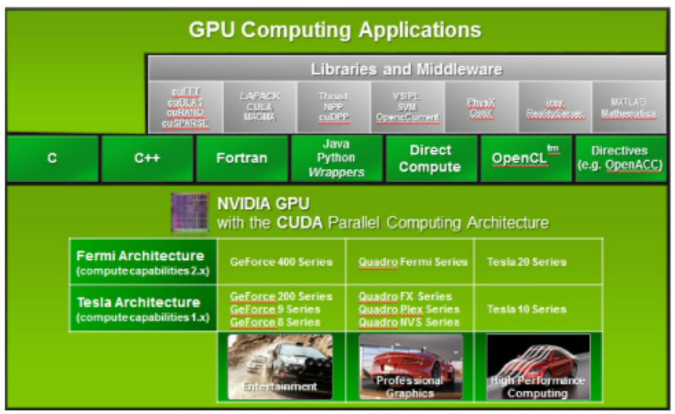
Compute Unified Device Architecture




GPU Computing Applications
Libraries and Middleware
cuFFT cuBLAS cuRAND cuSPARSE
LAPACK CULA MAGMA
Thrust NPP cuDPP
VSIPL SVM OpencCurrent
PhysX OptiX
iray RealityServer
MATLAB Mathematica
C
C++
Fortran
Java Python Wrappers
Direct Compute
OpenCL tm
Directives (e.g. OpenACC)
NVIDIA GPU
with the CUDA Parallel Computing Architecture
Fermi Architecture (compute capabilities 2.x)
GeForce 400 Series
Quadro Fermi Series
Tesla 20 Series
Tesla Architecture (compute capabilities 1.x)
GeForce 200 Series
GeForce 9 Series
GeForce 8 Series
Quadro FX Series
Quadro Plex Series
Quadro NVS Series
Tesla 10 Series
Entertainment
Professional Graphics
High Performance Computing

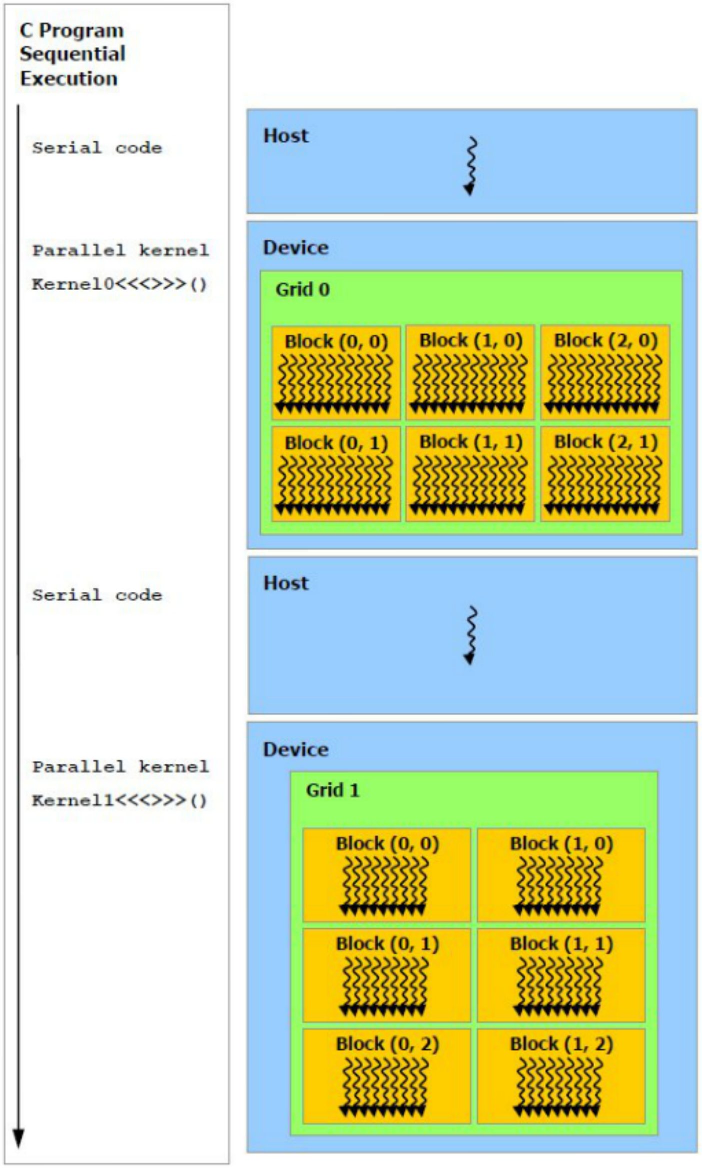
CUDA

Compute Unified Device Architecture



CUDA C : Execution model

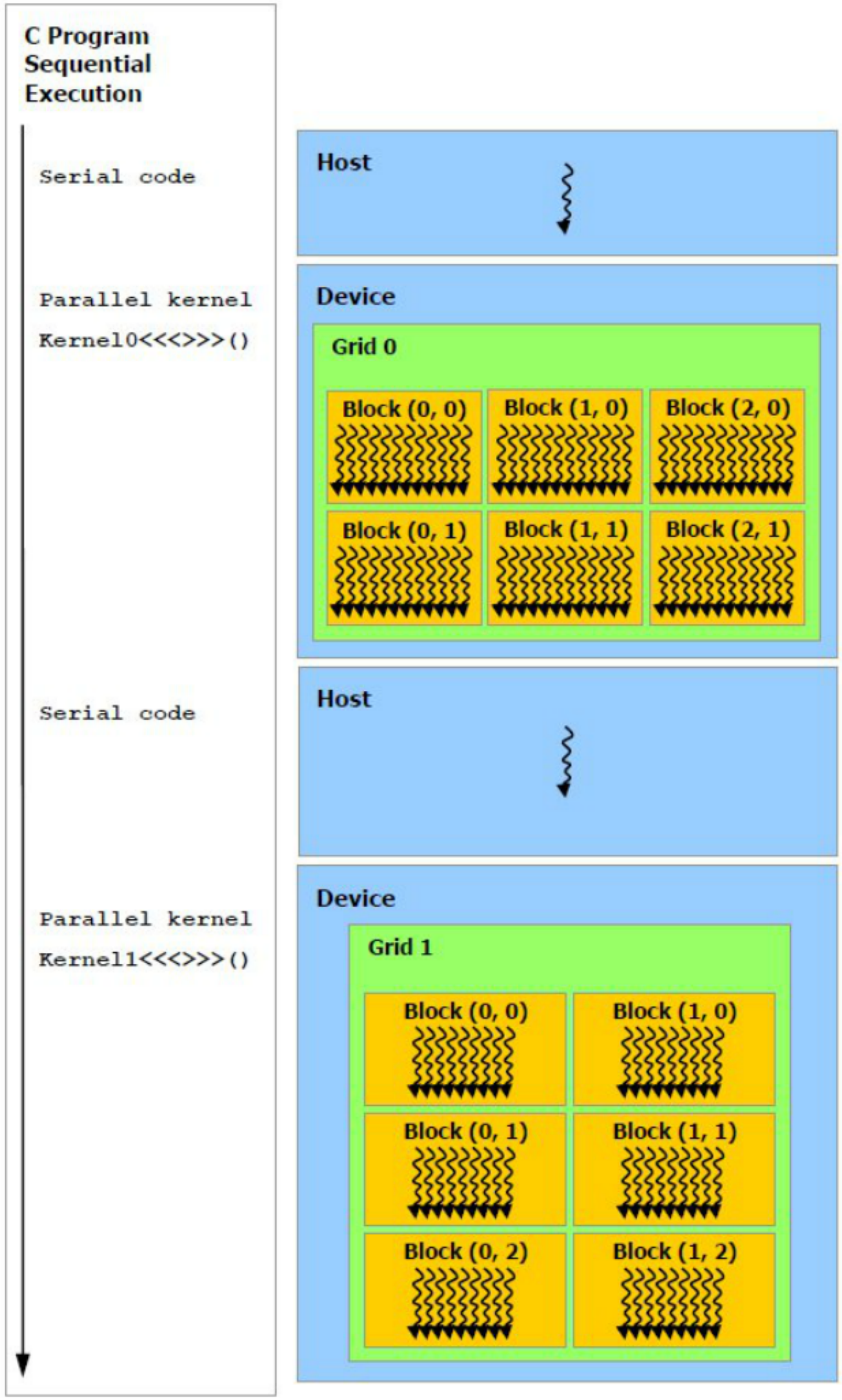
CUDA C = C + Some More keywords



Serial code executes on the host while parallel code executes on the device.

ution model

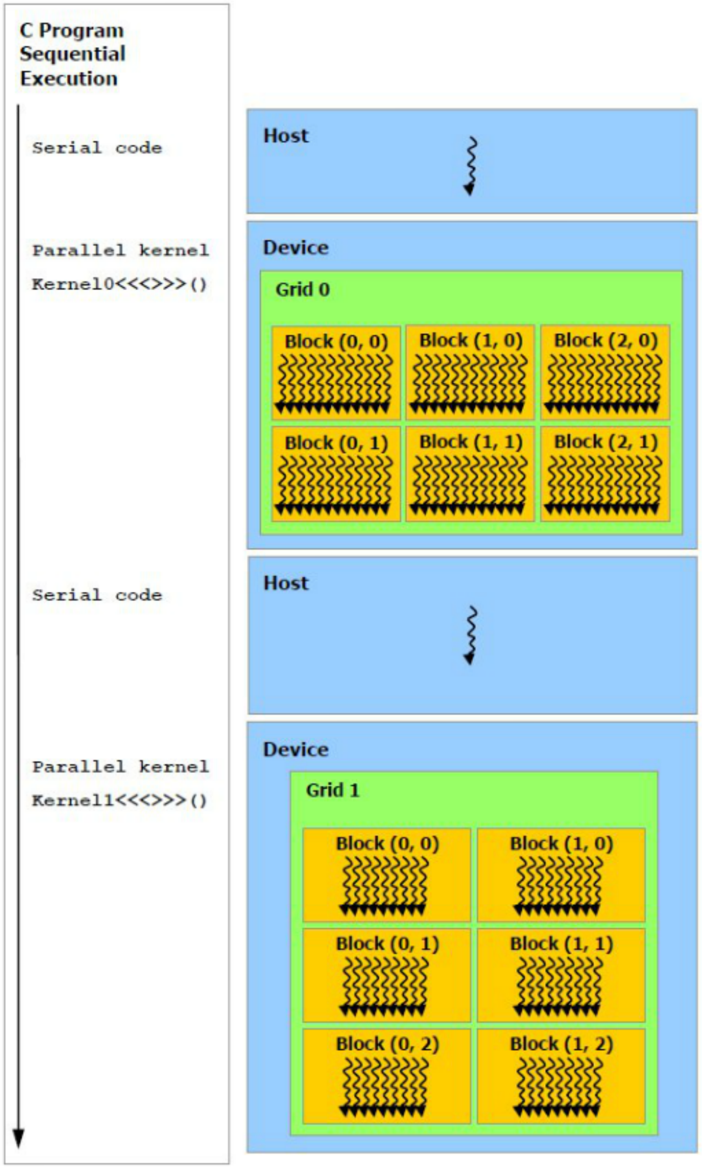
More keywords



Serial code executes on the host while parallel code executes on the device.

CUDA C : Execution model

CUDA C = C + Some More keywords



Serial code executes on the host while parallel code executes on the device.

*What Do I need to get started?***

- A CUDA-enabled graphics processor
- An NVIDIA device driver
- A CUDA development toolkit
- A standard C compiler

***Or you can ask Dr. Walsh to give you an account on aspitrg2*

Lets look at some code

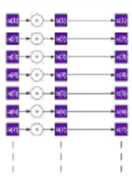
Hello World!

A Kernel Call

Passing Parameters

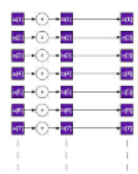
Know Your GPU(s)

```
printf( "   --- Memory Information for device %d ---\n", i );
    printf( "Total global mem:  %ld\n", prop.totalGlobalMem );
    printf( "Total constant Mem:  %ld\n", prop.totalConstMem );
    printf( "Max mem pitch:  %ld\n", prop.memPitch );
    printf( "Texture Alignment:  %ld\n", prop.textureAlignment );
    printf( "   --- MP Information for device %d ---\n", i );
    printf( "Multiprocessor count:  %d\n", prop.multiProcessorCount );
    printf( "Shared mem per mp:  %ld\n", prop.sharedMemPerBlock );
    printf( "Registers per mp:  %d\n", prop.regsPerBlock );
    printf( "Threads in warp:  %d\n", prop.warpSize );
    printf( "Max threads per block:  %d\n", prop.maxThreadsPerBlock );
    printf( "Max thread dimensions:  (%d, %d, %d)\n", prop.maxThreadsDim[0], prop.maxThreadsDim[1], prop.maxThreadsDim[2] );
    printf( "Max grid dimensions:  (%d, %d, %d)\n", prop.maxGridSize[0], prop.maxGridSize[1], prop.maxGridSize[2] );
    printf( "\n" );
}
```



Vector Addition

```
#include "book.h"
#define N 10
int main( void ) {
int a[N], b[N], c[N];
int *dev_a, *dev_b, *dev_c;
// allocate the memory on the GPU
HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );
// fill the arrays 'a' and 'b' on the CPU
for (int i=0; i<N; i++) {
a[i] = -i;
b[i] = i * i;
}
```



Vector Addition Contd...

```
// copy the arrays 'a' and 'b' to the GPU

  HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice ) );
  HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice ) );
  add<<<N,1>>>( dev_a, dev_b, dev_c );
  // copy the array 'c' back from the GPU to the CPU
  HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost ) );
  // display the results

  for (int i=0; i<N; i++) {
  printf( "%d + %d = %d\n", a[i], b[i], c[i] );
  }

  // free the memory allocated on the GPU
  HANDLE_ERROR( cudaFree( dev_a ) );
  HANDLE_ERROR( cudaFree( dev_b ) );
  HANDLE_ERROR( cudaFree( dev_c ) );
  return 0;
}
```

Hello World?

A Kernel Call

```
#include "../common/book.h"
_global__ void kernel( void ) {

}
int main( void ) {
    kernel<<<1,1>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

Passing Parameters

```
#include "book.h"
__device__ int addem( int a, int b ) {
   return a + b;
}

__global__ void add( int a, int b, int *c ) {
   *c = addem( a, b );
}

int main( void ) {

   int c;
   int *dev_c;

   HANDLE_ERROR( cudaMalloc( (void**)&dev_c,
sizeof(int) ) );
   add<<<1,1>>>( 2, 7, dev_c );
   HANDLE_ERROR( cudaMemcpy( &c, dev_c,
sizeof(int), cudaMemcpyDeviceToHost ) );

   printf( "2 + 7 = %d\n", c );
   HANDLE_ERROR( cudaFree( dev_c ) );
   return 0;

}
```

Know Your GPU(s)

```
#include "book.h"
int main( void ) {
  cudaDevicePropprop;

  int count;

  HANDLE_ERROR( cudaGetDeviceCount( &count ) );
  for (int i=0; i< count; i++) {
    HANDLE_ERROR( cudaGetDeviceProperties( &prop, i ) );
    printf( "   --- General Information for device %d ---\n", i );
    printf( "Name:  %s\n", prop.name );
    printf( "Compute capability:  %d.%d\n", prop.major,
prop.minor );
    printf( "Clock rate:  %d\n", prop.clockRate );
    printf( "Device copy overlap:  " );
    if (prop.deviceOverlap)
      printf( "Enabled\n" );
    else
      printf( "Disabled\n");
    printf( "Kernel execution timeout :  " );
    if (prop.kernelExecTimeoutEnabled)
      printf( "Enabled\n" );
    else
      printf( "Disabled\n" );

}
```

```
printf( "   --- Memory Information for device %d ---\n", i );
    printf( "Total global mem:  %ld\n", prop.totalGlobalMem );
    printf( "Total constant Mem:  %ld\n", prop.totalConstMem );
    printf( "Max mem pitch:  %ld\n", prop.memPitch );
    printf( "Texture Alignment:  %ld\n", prop.textureAlignment );
    printf( "   --- MP Information for device %d ---\n", i );
    printf( "Multiprocessor count:  %d\n", prop.multiProcessorCount );
    printf( "Shared mem per mp:  %ld\n", prop.sharedMemPerBlock );
    printf( "Registers per mp:  %d\n", prop.regsPerBlock );
    printf( "Threads in warp:  %d\n", prop.warpSize );
    printf( "Max threads per block:  %d\n", prop.maxThreadsPerBlock );
    printf( "Max thread dimensions:  (%d, %d, %d)\n", prop.maxThreadsDim[0],
    prop.maxThreadsDim[1], prop.maxThreadsDim[2] );
    printf( "Max grid dimensions:  (%d, %d, %d)\n", prop.maxGridSize[0],
    prop.maxGridSize[1], prop.maxGridSize[2] );
    printf( "\n" );
}
```

Lets look at some code

Hello World!

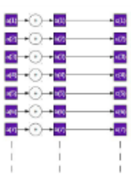
A Kernel Call

Passing Parameters

Know Your GPU(s)

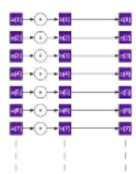
```
#include "book.h"
int main( void ) {
    cudaDeviceProp prop;

    int count;

    HANDLE_ERROR( cudaGetDeviceCount( &count ) );
    for (int i=0; i< count; i++) {
        HANDLE_ERROR( cudaGetDeviceProperties( &prop, i ) );
        printf( " --- General Information for device %d ---\n", i );
        printf( "Name: %s\n", prop.name );
        printf( "Compute capability: %d.%d\n", prop.major,
prop.minor );
        printf( "Clock rate: %d\n", prop.clockRate );
        printf( "Device copy overlap: " );
        if (prop.deviceOverlap)
            printf( "Enabled\n" );
        else
            printf( "Disabled\n");
        printf( "Kernel execution timeout : " );
        if (prop.kernelExecTimeoutEnabled)
            printf( "Enabled\n" );
        else
            printf( "Disabled\n" );

    printf( " --- Memory Information for device %d ---\n", i );
        printf( "Total global mem: %ld\n", prop.totalGlobalMem );
        printf( "Total constant Mem: %ld\n", prop.totalConstMem );
        printf( "Max mem pitch: %ld\n", prop.memPitch );
        printf( "Texture Alignment: %ld\n", prop.textureAlignment );
        printf( " --- MP Information for device %d ---\n", i );
        printf( "Multiprocessor count: %d\n", prop.multiProcessorCount );
        printf( "Shared mem per mp: %ld\n", prop.sharedMemPerBlock );
        printf( "Registers per mp: %d\n", prop.regsPerBlock );
        printf( "Threads in warp: %d\n", prop.warpSize );
        printf( "Max threads per block: %d\n", prop.maxThreadsPerBlock );
        printf( "Max thread dimensions: (%d, %d, %d)\n", prop.maxThreadsDim[0],
        prop.maxThreadsDim[1], prop.maxThreadsDim[2] );
        printf( "Max grid dimensions: (%d, %d, %d)\n", prop.maxGridSize[0],
        prop.maxGridSize[1], prop.maxGridSize[2] );
        printf( "\n" );
    }
}
```

Vector Addition



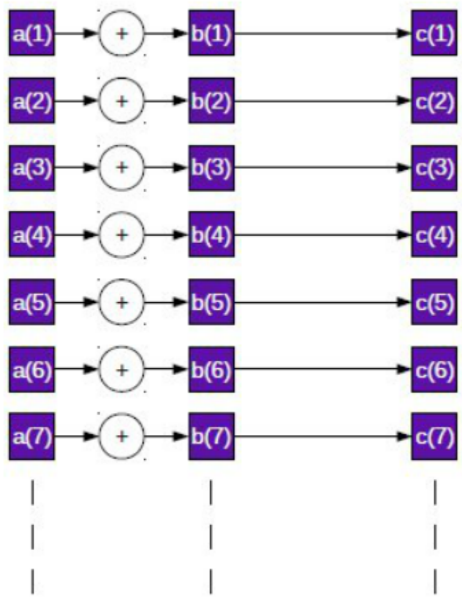
```
#include "book.h"
#define N 10
int main( void ) {
int a[N], b[N], c[N];
int *dev_a, *dev_b, *dev_c;
// allocate the memory on the GPU
HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );
// fill the arrays 'a' and 'b' on the CPU
for (int i=0; i<N; i++) {
a[i] = -i;
b[i] = i * i;
}
```

Vector Addition Contd…



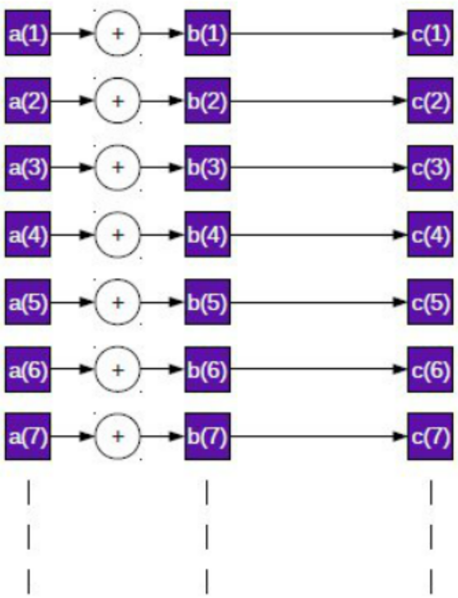
```
// copy the arrays 'a' and 'b' to the GPU

    HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice ) );
    add<<<N,1>>>( dev_a, dev_b, dev_c );
    // copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost ) );
    // display the results

    for (int i=0; i<N; i++) {
    printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }

    // free the memory allocated on the GPU
    HANDLE_ERROR( cudaFree( dev_a ) );
    HANDLE_ERROR( cudaFree( dev_b ) );
    HANDLE_ERROR( cudaFree( dev_c ) );
    return 0;
}
```

Vector Addition




```
#include "book.h"
#define N 10
int main( void ) {
int a[N], b[N], c[N];
int *dev_a, *dev_b, *dev_c;
// allocate the memory on the GPU
HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );
// fill the arrays 'a' and 'b' on the CPU
for (int i=0; i<N; i++) {
a[i] = -i;
b[i] = i * i;
}
```

Vector Addition Contd...




```
// copy the arrays 'a' and 'b' to the GPU

    HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice ) );
     add<<<N,1>>>( dev_a, dev_b, dev_c );
     // copy the array 'c' back from the GPU to the CPU
     HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost ) );
    // display the results

    for (int i=0; i<N; i++) {
    printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }

    // free the memory allocated on the GPU
    HANDLE_ERROR( cudaFree( dev_a ) );
    HANDLE_ERROR( cudaFree( dev_b ) );
    HANDLE_ERROR( cudaFree( dev_c ) );
    return 0;
}
```