Introduction to CUDA

T.V. Singh
Institute for Digital Research and Education
UCLA
tvsingh@ucla.edu

GPU and GPGPU

- Graphics Processing Unit
  - A specialized device on computer to accelerate building of images on Display
    - Graphics Accelerator
  - CUDA is developed by NVIDIA for General-purpose computing on GPU (GPGPU)
    - GPU as Computing Accelerator

Why GPUs?

- Low cost
- Low power consumption
- Existing market and user base
  - Most computer users already have systems with GPUs
- Primary focus of main vendors is not High Performance Computing (HPC)☺
- Necessary tools are free (well! mostly)

Why GPUs? - Performance

- NVIDIA’s Tesla K20x
  - 13 multiprocessors (x192 cores)= 2496 cores
  - 3.95 TFLOPS (SP) & 1.31 TFLOPS (DP)
  - 5 GB memory GDDR5
  - 208 GB/s memory bandwidth
- Tesla M2090
  - 16 multiprocessors (x32 cores)= 512 cores
  - 1.03 TFLOPS(SP) & 665 GFLOPS (DP_
  - 6 GB memory GDDR5
  - 177 GB/s memory bandwidth

Titan

- The 2nd most powerful supercomputer in Top500 list, June, 2013
  - 18,688 nodes with 16-core AMD Opteron 6274 CPU-2.2 GHz
  - 18,688 Nvidia Tesla K20X general purpose GPUs.
  - Theoretical peak 27 Pflops and 17.59 Pflops sustained.
  - 8.2 Mwatts power
- It replaced Jaguar
  - 18,688 compute nodes, each have two 6-cores processors-2.3 GHz
  - Theoretical peak 1381.4 Tflop/s and 1059 Tflops sustained
  - Same floor space as Titan
  - 6.950 Mwatts power
- Ref: http://www.olcf.ornl.gov/titan/

Tianhe-1A

- The 10th position supercomputer in Top500 list, June, 2013
  - 14,336 Xeon X5670
  - 7,168 Nvidia Tesla M2050 general purpose GPUs.
  - Theoretical peak 4.701 petaflops.
  - 4.04 Mwatts power
- In absence of GPUs, it would have taken
  - 50,000 Xeon X5670
  - Twice as much floor space to deliver the same performance
  - 12 Mwatts power
- Ref: http://en.wikipedia.org/wiki/Tianhe-IA

GPGPU in UCLA

- Started looking into exotic architectures in 2007
- First lunch meeting in 2007 on GPGPU
- Few projects/case studies started around the campus
- GPU workshop in July 2009
- MRI proposal for GPU based cluster – funded $1.78M
- groups.google.com/group/ucla-gpu
- Classes for CUDA and/or other topics in GPGPU

GPGPU in UCLA- Hoffman2

- 7 node Tesla setup
    - Dual Intel Xeon X5560 @2.6 GHz
    - Memory: 24 GB
    - Network: Infiniband
    - 4 nodes, each with 2 Tesla S1070
    - 2 nodes, each with 2 Tesla S2050
    - 1 node, with 6 Tesla M20
    - For access: www.idre.ucla.edu/hoffman2

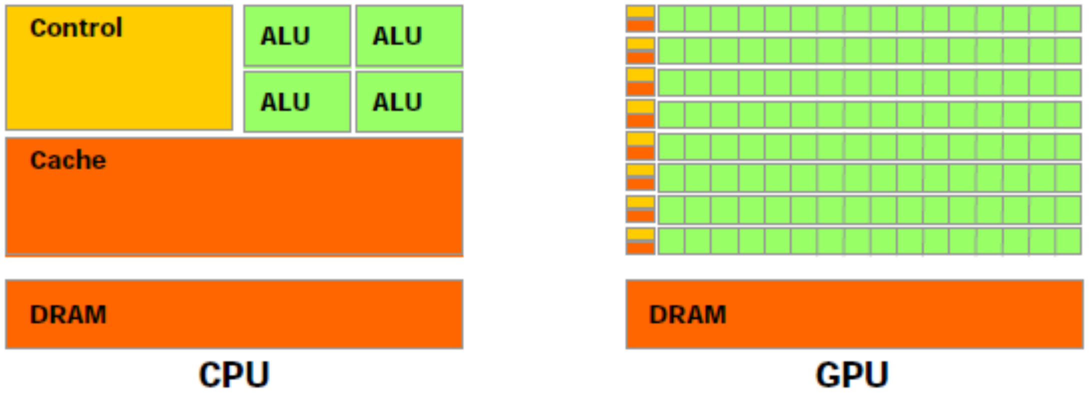
GPGPU in UCLA- Dawson2

- GPUs based cluster
    - 96x12 processors
    - 96x3 NVIDIA Tesla 2090 GPUs
    - 96x48 GB total host memory
    - Top 500 listing:
        - 148 in Jun 2011
        - 234 in Nov 2011
        - 384 in Jun 2012

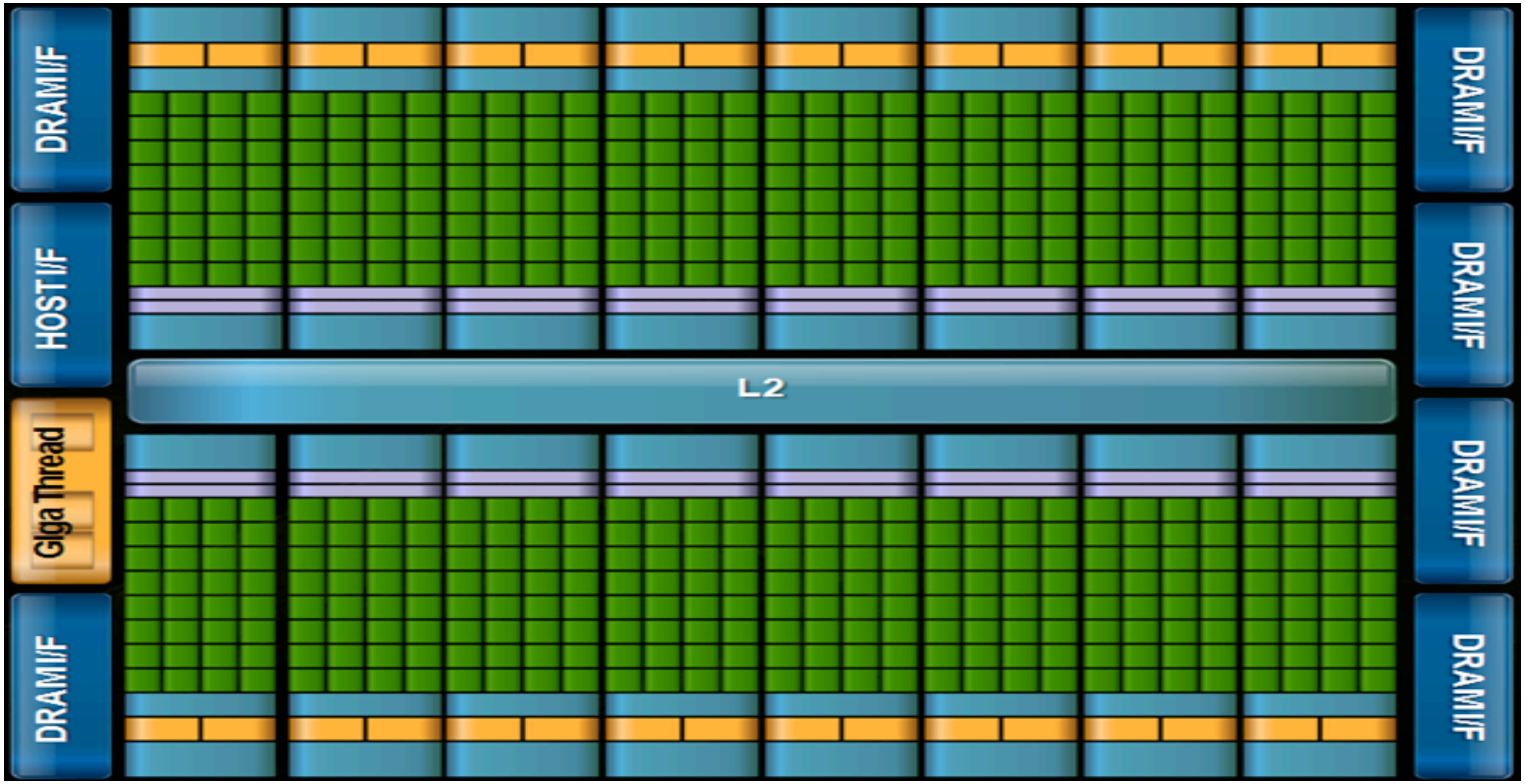
Hoffman2 vs Dawson2 cluster @UCLA

- Hoffman2 (as of July 28th 2013)
  - 1033 nodes or 9783 processors
  - 102.8 TFLOPS peak
  - Power ~200 KW
- Dawson2
  - 96 nodes or 1152 processors
    - Plus 288 GPUs (Tesla M2090)
  - ~200 TFLOPS peak
    - Sustained 70.28 TF, positioned at 384 in June 2012's top500 list
  - Power 96 KW
    - 731 MFLOPS/W, listed at 40th in Nov. 2011's green500 list

CPU vs GPU

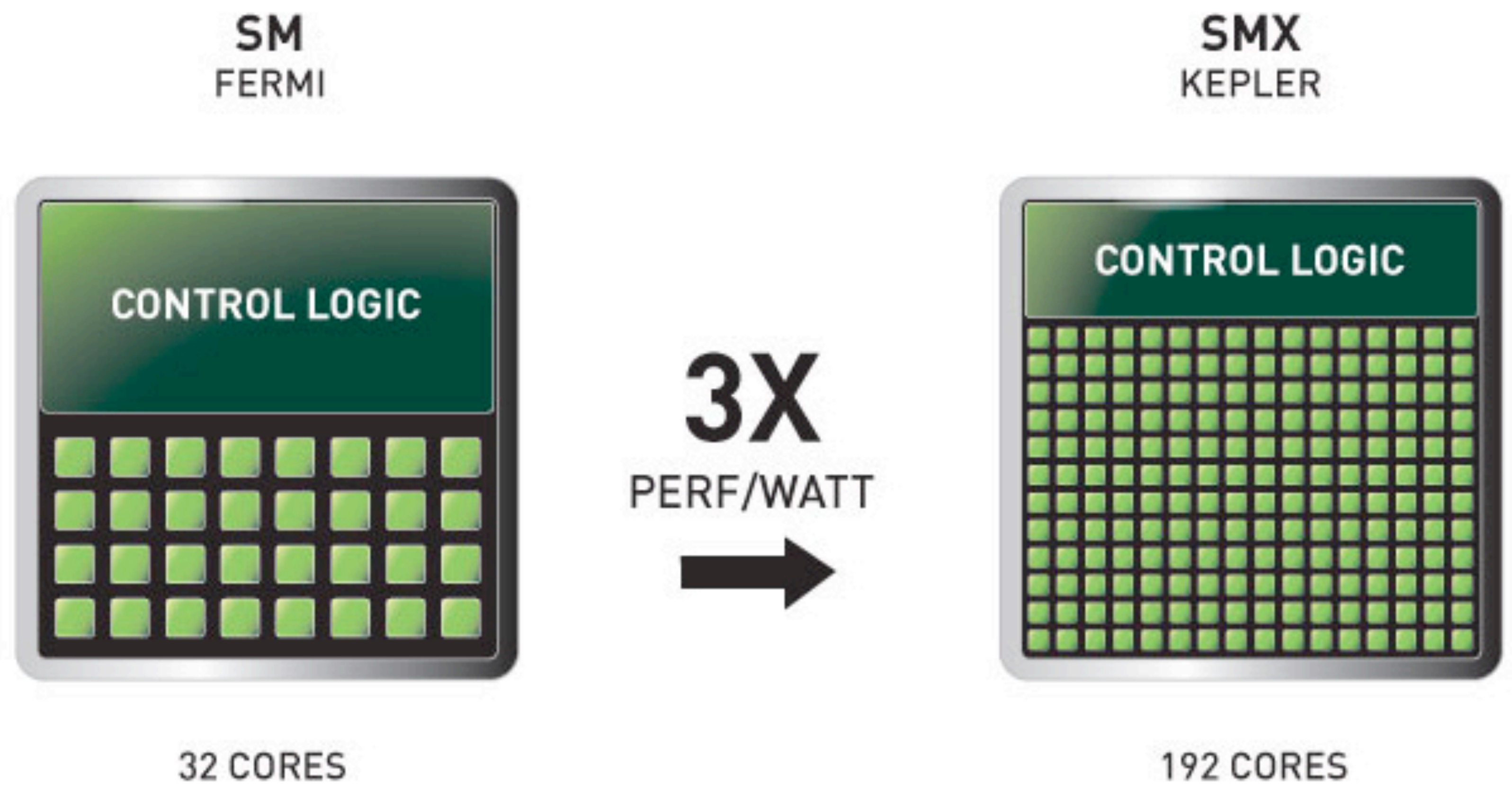


Fermi Architecture-M2090



Source: Nvidia

Kepler Architecture- K20 & K20X

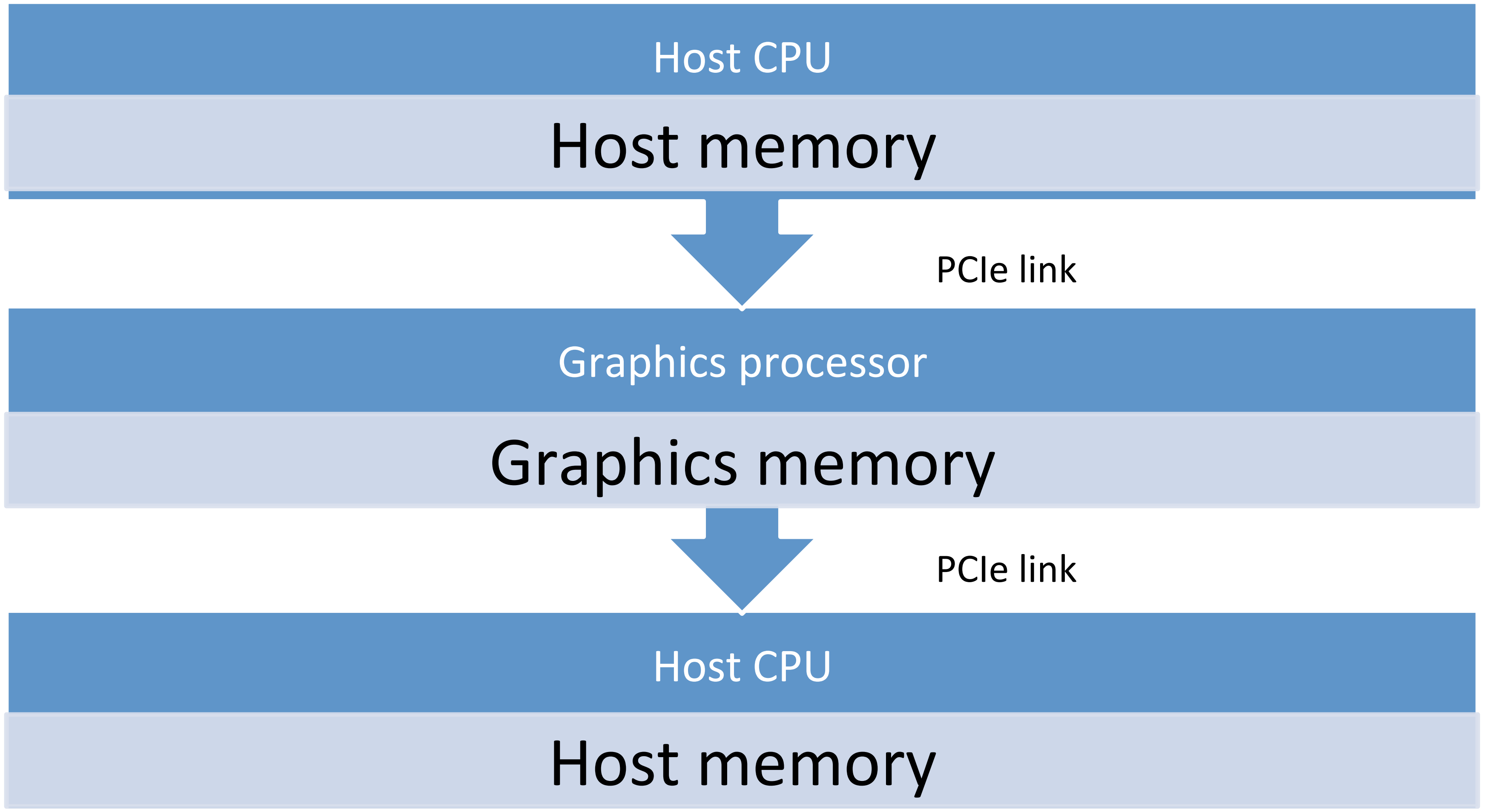


Source: Nvidia

CUDA

- Compute Unified Device Architecture
  - From NVIDIA
  - GP parallel computing architecture
    - CUDA-Instruction set architecture (ISA)
    - Parallel compute engine in GPU
  - Associated software provides
    - Small set of extensions to C
    - Supports heterogeneous computing on host and GPU

General purpose computation on graphics hardware



Getting started with CUDA

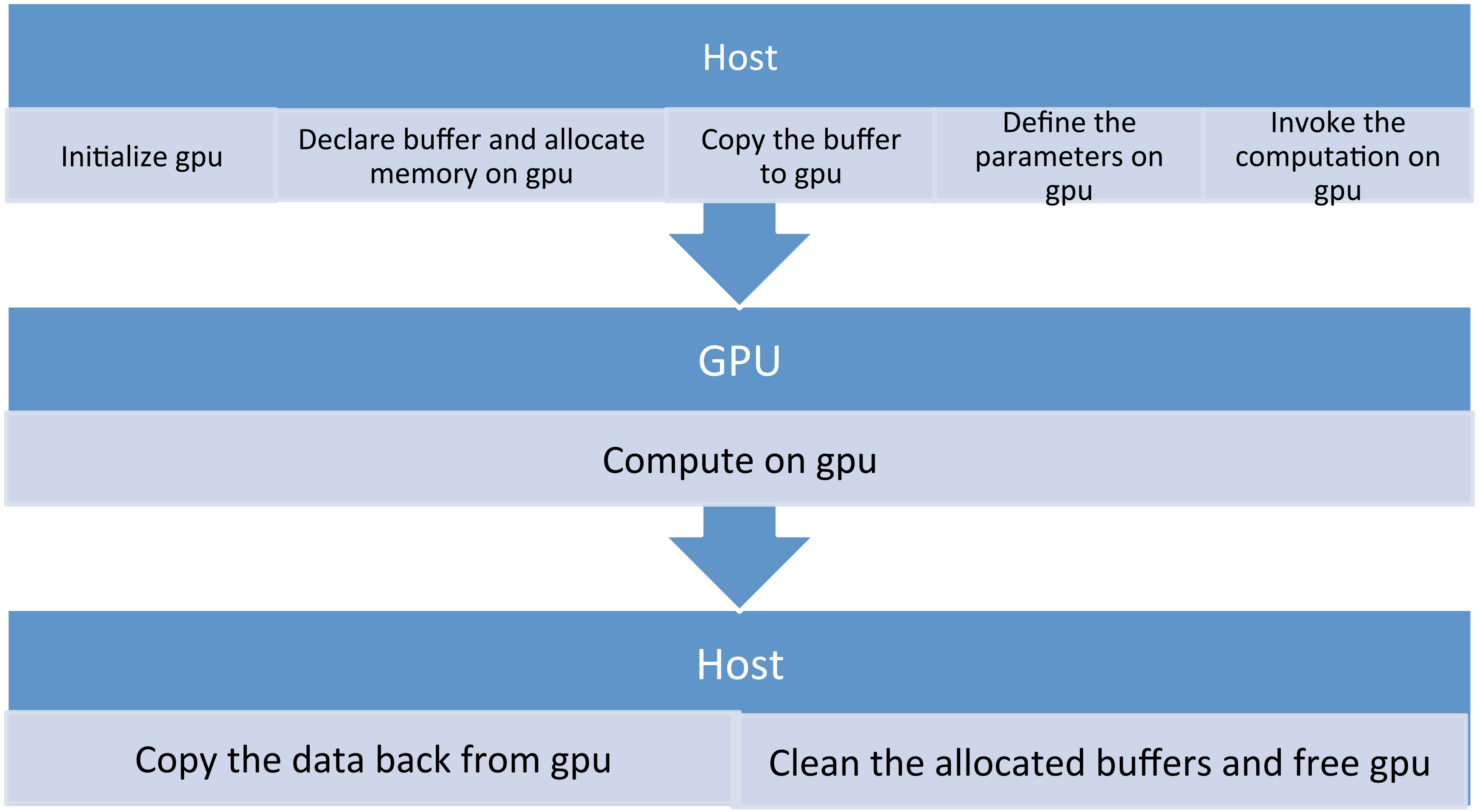
- What you need
  - A system (Windows, Mac or Linux)
  - CUDA enabled GPU (from NVIDIA)
  - CUDA software development kit
- Install and verify the installation of CUDA and GPU device

*More on http://www.nvidia.com/cuda

C-CUDA

- Provides a simple path for users familiar with C language
- Consists a minimal set of extensions to C Language
- Contains a run time library cudart
- Cudart gets initialized the first time any cuda runtime function is called
- On systems, with multiple GPUs, the code runs on device 0 or first GPU by default

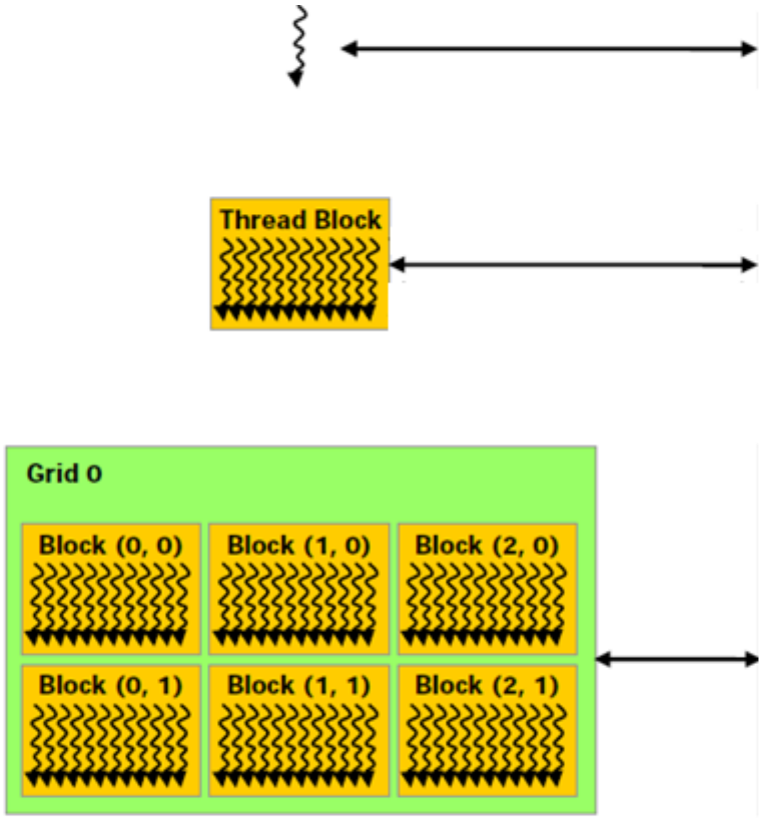
General purpose computation on graphics hardware


Host
Initialize gpu
Declare buffer and allocate memory on gpu
Copy the buffer to gpu
Define the parameters on gpu
Invoke the computation on gpu
GPU
Compute on gpu
Host
Copy the data back from gpu
Clean the allocated buffers and free gpu

Simple C-CUDA Program on GPU

```
#include stdlib.h>
#include "cuda.h"

__global__ void zeroKernel() {}

main() {
  dim3 dimBlock(1);
  dim3 dimGrid(1);
  cudaError_t crc;

  zeroKernel<<<dimGrid,dimBlock>>>();

  cudaThreadSynchronize();

  crc = cudaGetLastError();

  if (crc) {
    printf(”zeroKernel error=%d:%s\n",crc,cudaGetErrorString(crc));
    exit(1);
  }
}
```

Thread hierarchy

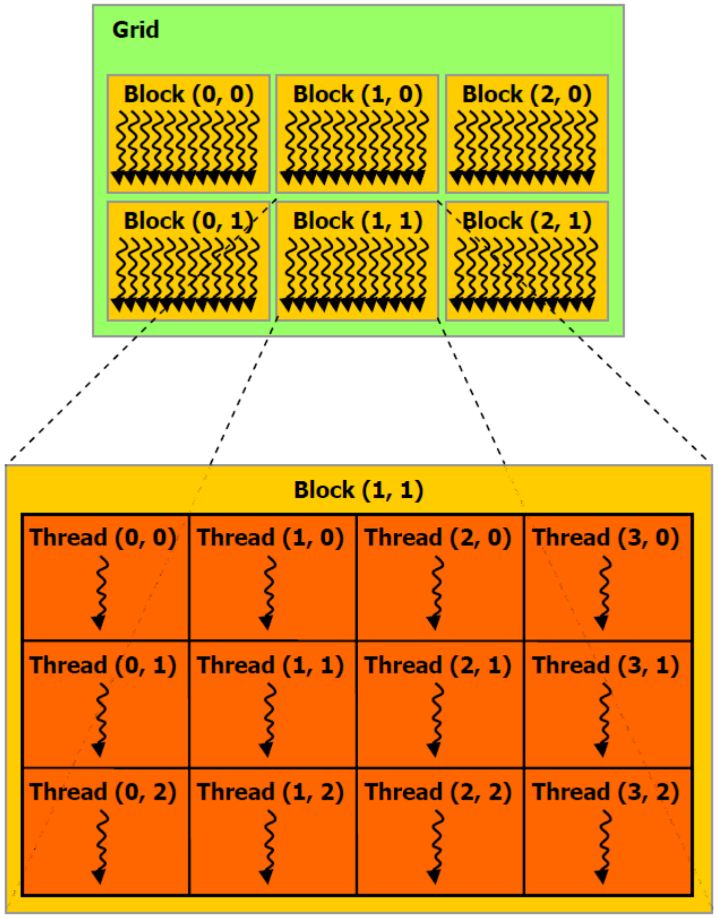


Thread

Block of Threads

Grid of Blocks

Thread hierarchy



Thread hierarchy..

- Threads
  - Identified by threadIdx
  - One, two or three dimensions
    - Thread block (Dx,Dy)
    - Thread ID of index (x,y) = x+y*Dx
    - Thread block (Dx,Dy,Dz)
      - Thread ID of index (x,y,z) = x+y*Dx+z*Dx*Dy
- Blocks
    - Identified by blockIdx in a Grid
    - One, two or three dimensions possible

CUDA programming model

- Kernel
  - Invoked using modified C function call
  - Runs on GPU
  - Runs in parallel as CUDA threads
  - Function defined using __global__ type
  - Each kernel gets a unique id i.e. thredIdx
  - Number of threads are defined by host program
  - Initiated by host program

Kernel syntax

```
__global__ void zeroKernel() {}

int main(){
   dim3 dimBlock(256);  dim3 dimGrid(256,256);
   zeroKernel<<<dimGrid,dimBlock>>>();}
```

Or a real example as:

```
__global__ void VecAdd(float* A, float* B, float* C)
{
   int i = blockIdx.x*blockDim.x+threadIdx.x;
   if(i < N) C[i] = A[i] + B[i];
}
int main()
{
  ...
  // Kernel invocation
  dim3 dimBlock(block_size);
  dim3 dimGrid(N/dimBlock.x);
  VecAdd<<<dimGrid, dimBlock>>>(A, B, C);
   cudaThreadSynchronize();
}
```

Kernel and some variables

- __global__
  - __device__
- dim3 dimBlock(block_size);
  - dim3 dimBlock(n1,n2,n3);
  - n1*n2*n3 <=1024
  - dim3 threadIdx- thread index within block
- dim3 dimGrid(N/dimBlock.x);
  - dim3 dimGrid(n1,n2,n3);
  - n1, n2, n3 <= 65535 for C2050/C2090, n1 for K20 <= 2147483647
  - dim3 blockIdx – block index within grid
- VecAdd<<<dimGrid, dimBlock>>>(A, B, C);
  - VecAdd<<<dimGrid, dimBlock, size_t bytes, stream id>>>(A, B, C);
- cudaThreadSynchronize();

Example-vector addition

```
#include<stdio.h>
# define N 32

main()
{
    float a[N], b[N], c[N];
    int i;

    for (i=0; i< N; i++){  a[i] = 1.0*I; b[i] = 1.0*i*i; } //for initialization

    for(j=0;j<N;j++) C[j]=A[j]+B[j];
}
```

Compute intensive part

```
#include<stdio.h>
# define N 32
 void VecAdd(float* A, float* B, float* C){
       int j;
       for(j=0;j<N;j++) C[j]=A[j]+B[j];
       return;
}
main(){
       float a[N], b[N], c[N];
       int I;
        for (i=0; i< N; i++){  a[i] = 1.0*I; b[i] = 1.0*i*i; }
         VecAdd(a,b,c);
}
```

VectorAdd function for GPU

- On host:
  - for(j=0;j<N;j++) C[j]=A[j]+B[j];
- Or we can write it as:
  - j=0;
  - if(j<N){C[j]=A[j]+B[j]; j++;}
- On GPU, think each index j is an index for a thread. Meaning j is predefined as:
  - j=blockIdx.x*blockDim.x+threadIdx.x;
  - So the code reduces to just:
    - If(j<N)C[j]=A[j]+B[j];

VectorAdd function for GPU

```
__global__ void VecAdd(float* A, float* B, float* C)
{
unsigned int j   = blockIdx.x*blockDim.x+threadIdx.x;
if(j < N) C[j]=A[j]+B[j];
return;
}
```

- Represent single thread
- J - determined through threadId
- Function is declared as global

Main's modification for GPU

```
main()
{
float a[N], b[N], c[N];
int i;
for (i=0; i< N; i++){  a[i] = 1.0*I; b[i] = 1.0*i*i; }

 float *a_d, *b_d, *c_d;                    /* Declare the GPU buffers  */

cudaMalloc ((void **) &a_d , sizeof(float)*N);  ┐
cudaMalloc ((void **) &b_d , sizeof(float)*N);  ├     /* Allocate memory on the device */
cudaMalloc ((void **) &c_d , sizeof(float)*N);  ┘
        cudaMemcpy( a_d, a,  sizeof(float)*N   ,cudaMemcpyHostToDevice);  ┐     /* Copy data from host to device */
        cudaMemcpy( b_d, b,  sizeof(float)*N   ,cudaMemcpyHostToDevice);  ┘
dim3 dimBlock(BLOCK_SIZE);                ┐
dim3 dimGrid (N/dimBlock.x);              ├ /* Define Grid and blocks of threads */
 if( N % BLOCK_SIZE != 0 ) dimGrid.x+=1;  ┘
        VecAdd<<<dimGrid,dimBlock>>>(a_d,b_d,c_d);  ┐     /* Invoke the kernel and wait to finish */
        cudaThreadSynchronize();                    ┘
  cudaMemcpy( c, c_d,  sizeof(float)*N   ,cudaMemcpyDeviceToHost);       /* Copy the data to host */
  cudaFree(a_d);  cudaFree(b_d); cudaFree(c_d);
}
```

Memory management functions

- cudaMalloc ((void **) &a_d, sizeof(float)*N);
  - Allocates the memory on device.
- cudaFree(a_d);
  - Free the variable pointer a_d on device.

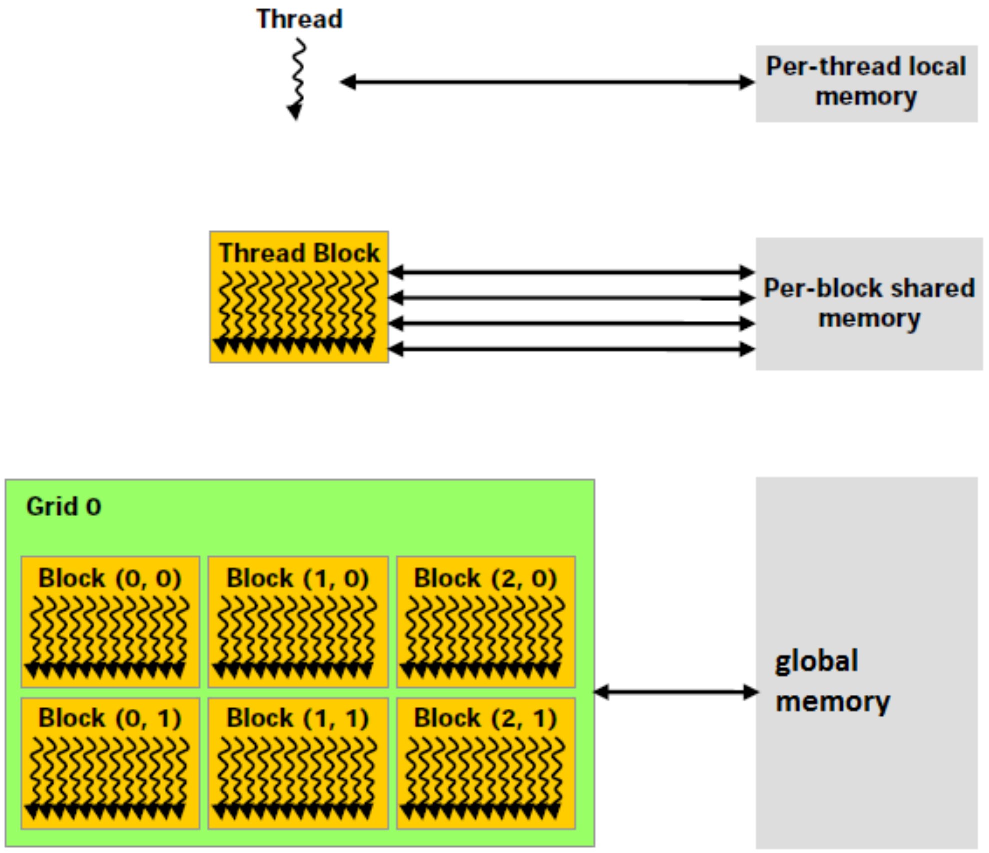
- cudaMemcpy(a_d, a, sizeof(float)*N, cudaMemcpyHostToDevice);
  - Copies the data from host to device.
- cudaMemcpy(a, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
  - Copies the data from device to host.

  - Enum CudaMemcpyKind
    - cudaMemcpyDeviceToHost
    - cudaMemcpyHostToDevice
    - cudaMemcpyDeviceToDevice

Compiling and running

- Compilation
  - nvcc vector_add.cu –o vector_add
- Running
  - ./vector_add
- NVCC
  - File with cuda sources must be compiled with nvcc
  - nvcc is compiler driver and invokes other necessary tools for the purpose of compilation
  - Generates CPU code or PTX (for GPU) code
  - Executable requires CUDA runtime libraries (-lcudart)

Memory hierarchy



Shared memory

- Maximum size is 48K bytes on C2050.
- On chip -very low latency and very high bandwidth
- Like a per-multiprocessor based cache if no bank conflicts
- Much faster than the global memory

- Declared either using
  - __shared__ __device__ a_d[N]
- Or
  - extern __shared__ a_d;
    - Size specified by third parameter in kernel call
    - Vecadd<<<dimGrid,dimBlock,shared_size,stream>>>(….);

CUDA warp

- CUDA utilizes SIMT (single instruction multiple threads)
- SIMT unit creates, manages, schedules and executes threads in group of 32 parallel threads.
- The groups of these 32 threads are warps.
- Half warp is either first or later half of a warp.
- Individual threads in a warp start together but are otherwise free to execute independently.

Memory coalescing

- Global memory access by a half warp coalesced in to a single transaction if:
  - It accesses contiguous region of global memory:
    - 64 bytes - each thread reads 4 bytes.
    - 128 bytes - each thread reads 8 bytes
    - 256 bytes – each thread reads 16 bytes
- This defines how the optimum memory bandwidth can be achieved on GPU

Example-Common way of Transpose

```
#include <stdlib.h>
#include <stdio.h>
#include "cuda.h“
#define BLOCK_DIM  16

__global__ void transpose(float *odata, float
      *idata, int W, int H)
{
  unsigned int xIndex = blockDim.x * blockIdx.x +
      threadIdx.x;
  unsigned int yIndex = blockDim.y * blockIdx.y +
      threadIdx.y;

  if (xIndex < width && yIndex < height)
  {
     unsigned int index_in  = xIndex + W * yIndex;
     unsigned int index_out = yIndex + H * xIndex;
     odata[index_out] = idata[index_in];
  }
```

```
main () {
 int  i,j, M=16, N=16;
 float a[M*N], at[M*N];
          for (j=0; j< M*N; j++) a[j] = rand()%100;
float *a_d, *at_d;
cudaMalloc((void **)&a_d , M*N*sizeof(float));
cudaMalloc((void **)&at_d , M*N*sizeof(float));

cudaMemcpy( a_d, a,  M*N*sizeof(float),
      cudaMemcpyHostToDevice);

dim3 dimBlock(BLOCK_DIM,BLOCK_DIM,1);
dim3 dimGrid(M/BLOCK_DIM,N/BLOCK_DIM);

transpose<<<dimGrid,dimBlock>>>(at_d,a_d,M,N);

cudaThreadSynchronize();
cudaMemcpy(at, at_d, M*N*sizeof(float),
      cudaMemcpyDeviceToHost);

  cudaFree(a_d);  cudaFree(at_d);}
```

Example-With shared memory

```
#include <stdlib.h>
#include <stdio.h>
#include "cuda.h“
#define BLOCK_DIM  16

__global__ void transpose(float *odata, float *idata, int W, int H)
{
        __shared__ float block[BLOCK_DIM][BLOCK_DIM+1];

        // read the matrix tile into shared memory
        unsigned int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
        unsigned int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;

        if((xIndex < W) && (yIndex < H)){
                unsigned int index_in = yIndex * W + xIndex;
                block[threadIdx.y][threadIdx.x] = idata[index_in];}

        __syncthreads();

        // write the transposed matrix tile to global memory
        xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
        yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;

        if((xIndex < H) && (yIndex < W)){
                unsigned int index_out = yIndex * H + xIndex;
                odata[index_out] = block[threadIdx.x][threadIdx.y];}
}
```

GPU Device management

- GPU Device query
  - cudaChooseDevice(int*devNumber, cudaDeviceProp* prop)
  - cudaGetDevice(int* devNumber)
  - cudaGetDeviceCount(int *devCount)
  - cudaGetDeviceProperties(cudaDeviceProp* prop, int devNumber)
  - cudaSetDevice(int devNumber)
- Important
  - GPU Device 0 is used by default

Using Hoffman2 for GPGPU

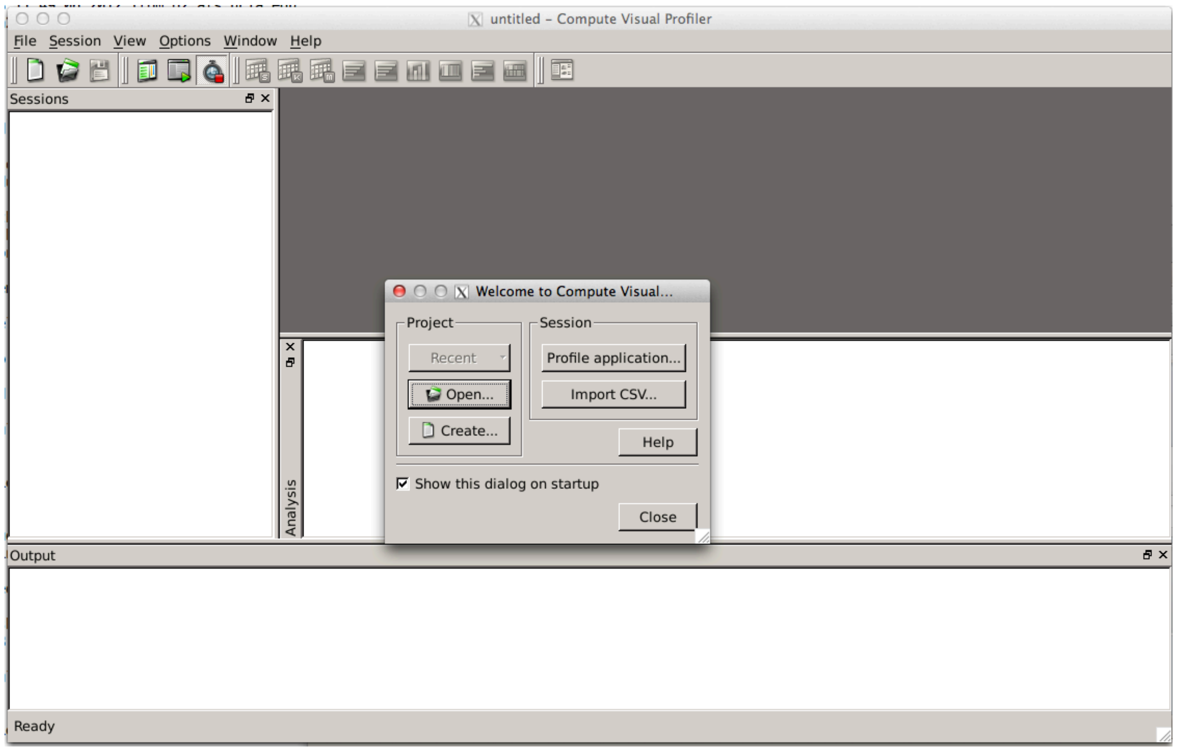
- Apply through following link for add resources at page:
    - https://idre.ucla.edu/hoffman2/getting-started#newuser
- On Hoffman2
    - qrsh –l i,gpu   (for access to interactive gpu nodes)
    - qrsh –l gpu (for access to batch only gpu nodes)
    - module load cuda

CUDA debugger

- Use nvcc with
  - –g –G for compilation
  - Optional: either of the following (depending on architecture of GPU)
    - -gencode arch=compute_20,code=sm_20
    - -gencode arch=compute_10,code=sm_10
- Command for debugger is cuda-gdb
  - Available through module cuda

CUDA Profiler

- Invoke command computeprof
  - Launch application
    - Graphical interface for profiling the code


untitled – Compute Visual Profiler
File Session View Options Window Help
Sessions
Analysis
Welcome to Compute Visual...
Project
Recent
Open...
Create...
Session
Profile application...
Import CSV...
Help
Show this dialog on startup
Close
Output
Ready

UCLA IDRE

CUDA Libraries

- CUDA provides many libraries:
  - CUBLAS
    - CUDA BLAS library
  - CUFFT
    - Provides a simple interface for computing FFTs using NVIDIA’s CUDA enabled GPUs.
  - NPP
    - For imaging and video processing
  - CUSPARSE
    - GPU-accelerated sparse matrix library
  - CURAND
    - GPU-accelerated RNG library

Alternatives- OpenACC Initiative

- Set of directives
- For C,C++ and Fortran
- Objectives include providing portability across Oss, host CPUs and accelerators
- Current partners: CRAY, NVIDIA, CAPS and Cray
- http://www.openacc-standard.org/

Fortran accelerator from Portland group

- CUDA extensions to Fortran 90/95
- Developed by PGI
- Use CUDA under the hood

Alternatives- OpenCL

- Open computing language
- Initiated by Apple
- Maintained by Khronos group
- Similar to CUDA driver
- Supported by most vendors e.g Intel, Nvidia, AMD, IBM and Apple

For further discussion

tvsingh@ucla.edu

For further study

www.nvidia.com/cuda
http://www.khronos.org/opencl/
www.gpgpu.org
http://www.pgroup.com/resources/cudafortran.htm

In particular
NVIDIA CUDA Programming guide
CUDA Best Practices Guide

UCLA IDRE