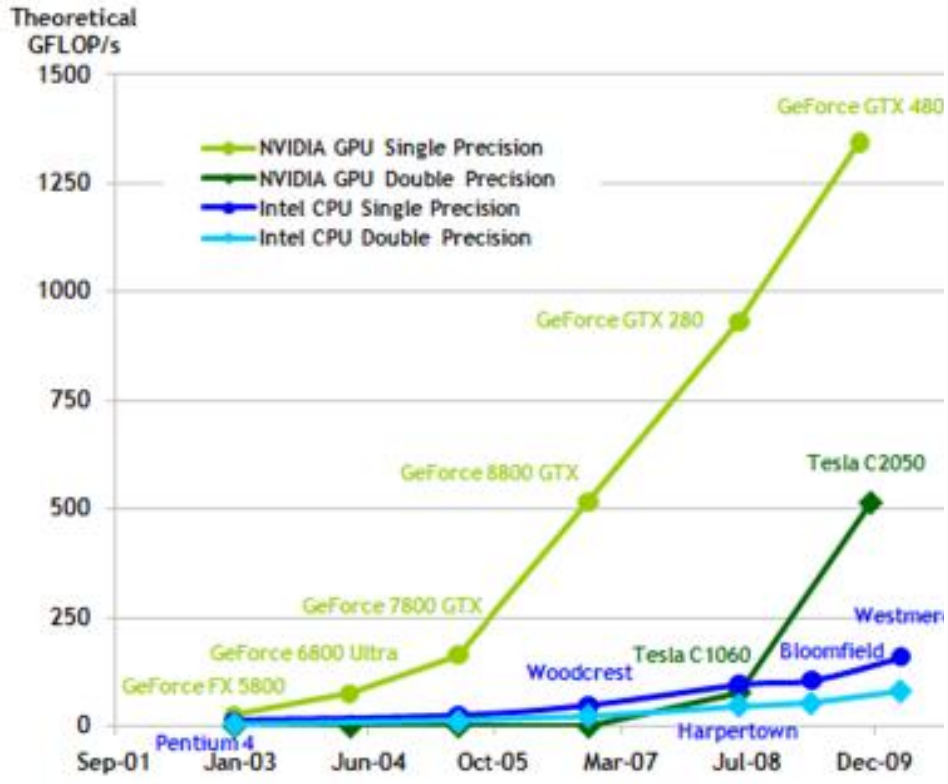
CUDA BASICS

Matt Heavner -- mheavner@buffalo.edu
10/21/2010

Why CUDA?



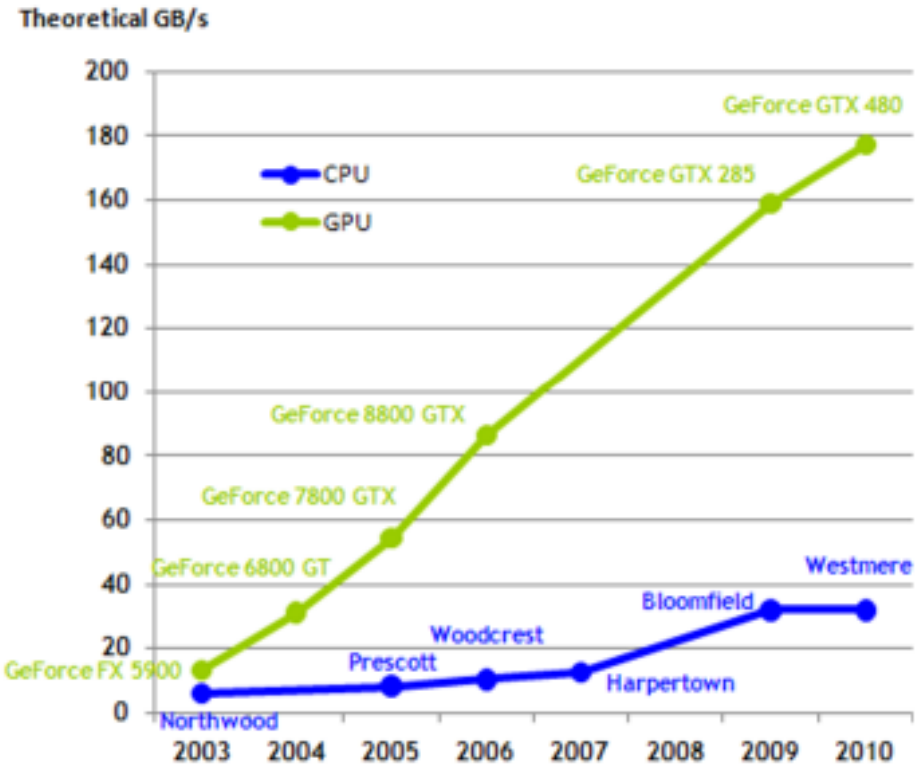


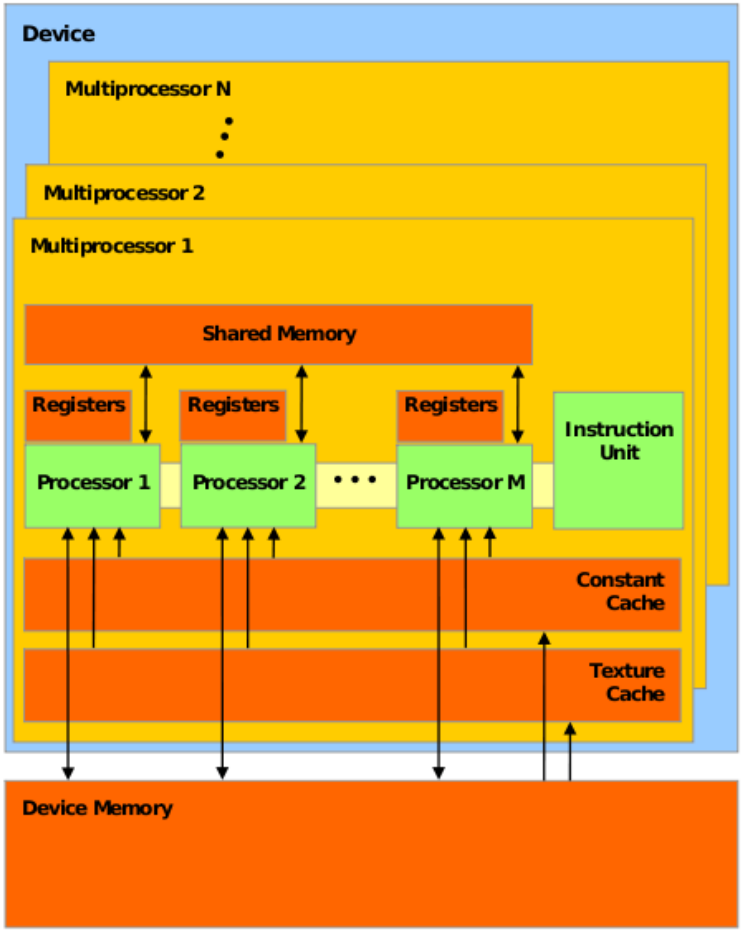
Image From: http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf

CUDA: Where does it fit into a problem?

- A “SIMD” architecture
- Works well when a similar operation is applied to a large dataset
  - Can also branch off, though, so not strictly SIMD
  - Actually “SIMT”
- Provides a small amount of additional syntax to C or C++ which allows parallel “kernels” to be run on the device

CUDA Physical Architecture

- Build around Streaming Multiprocessors (SMs)
- Each SM has 8 processing cores
- Each thread mapped to one SM
- Threads managed in groups of 32 – warps (basically, a SIMD group)
- Warp elements free to branch, though device will then serialize



A set of SIMT multiprocessors with on-chip shared memory.

Figure 4-2. Hardware Model

Image From: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf

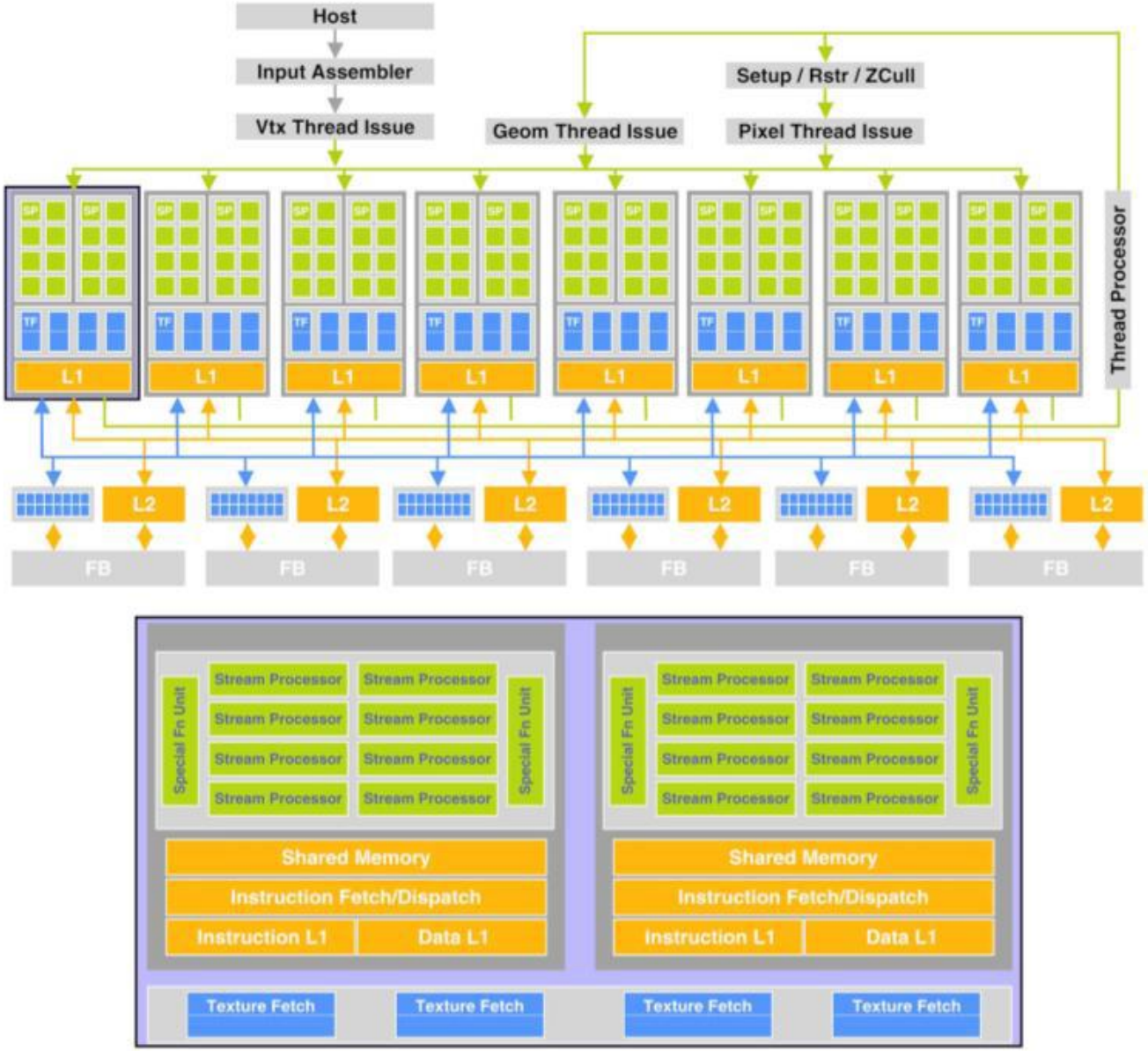


Fig. 1. *Today, both AMD and NVIDIA build architectures with unified, massively parallel programmable units at their cores. (a) The NVIDIA GeForce 8800 GTX (top) features 16 streaming multiprocessors of 8 thread (stream) processors each. One pair of streaming multiprocessors is shown below; each contains shared instruction and data caches, control logic, a 16 kB shared memory, eight stream processors, and two special function units. (Diagram courtesy of NVIDIA.)*

Magic Compute Capability

- 13 nodes
- 4 Tesla S1070 Units / Node
- 240 Streaming Processors / S1070
- → 12,480 total CUDA cores

Image From: http://www.tomshw.it/files/2009/12/immagini/23228/tesla-s1070_t.jpg

CUDA Compute Capability

- CUDA products are divided into compute capabilities 1.0, 1.1, 1.2, 1.3, 2.0
- The architecture present on magic has compute capability 1.3
  - 1.2/1.3 adds floating point support
  - Max active warps / multiprocessor = 32
  - Max active threads / multiprocessor = 1024
  - Most flexibility

CUDA Kernels

- A kernel is the piece of code executed on the CUDA device by a single CUDA thread
- Each kernel is run in a thread
- Threads are grouped into warps of 32 threads. Warps are grouped into thread blocks. Thread blocks are grouped into grids.
- Blocks and grids may be 1d, 2d or 3d
- Each kernel has access to certain variables that define its position – gridDim, blockIdx, blockDim, threadIdx. Useful for indexing datasets
- While host code may be C++, kernel must be in C along with CUDA syntax extensions

CUDA Logical Architecture

- Threads are grouped into thread blocks. Thread blocks are grouped into grids.

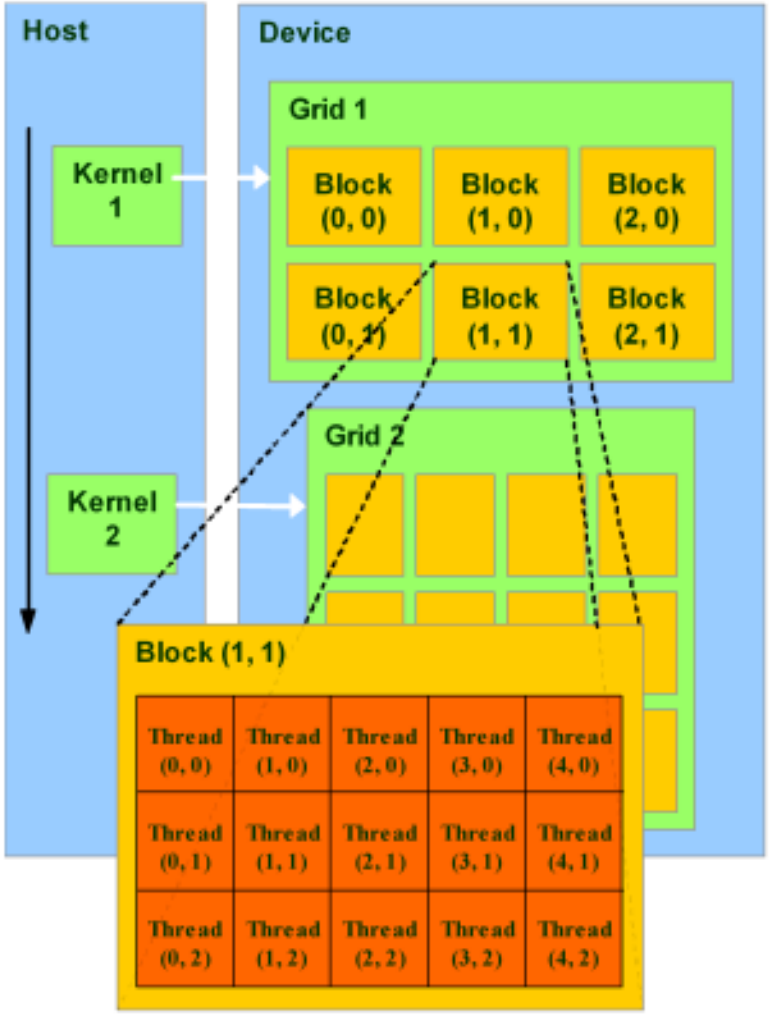


Image From: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf

CUDA Logical Architecture

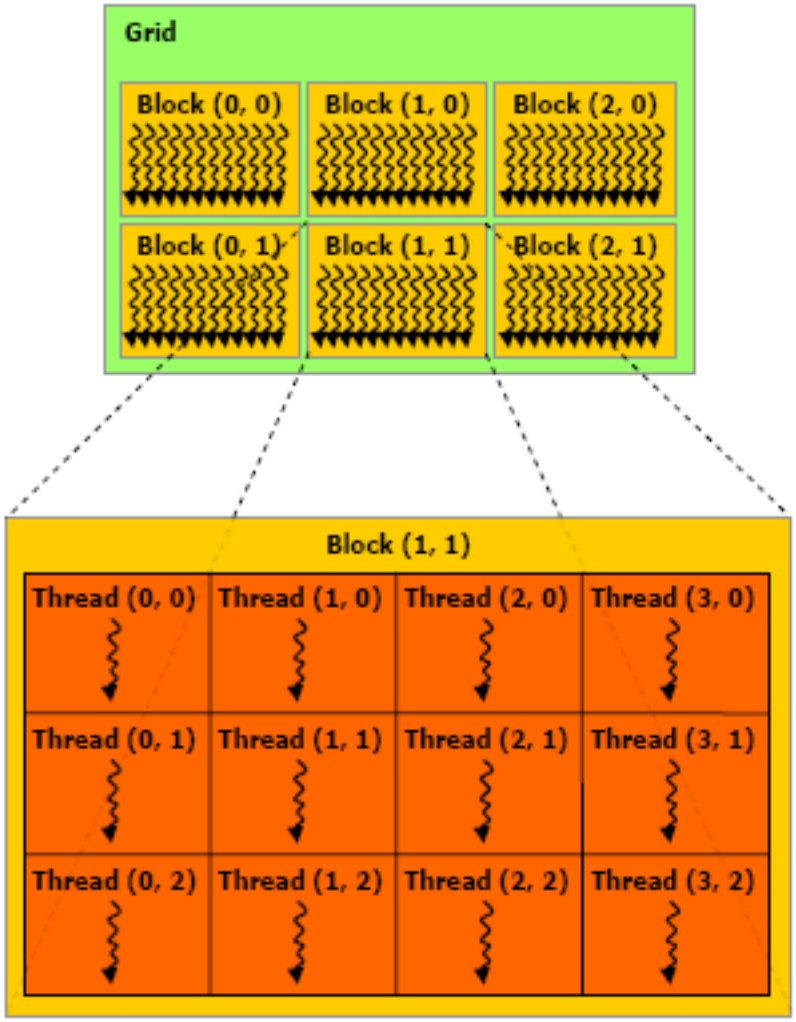


Image From: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf

CUDA Logical Architecture -- Scalability

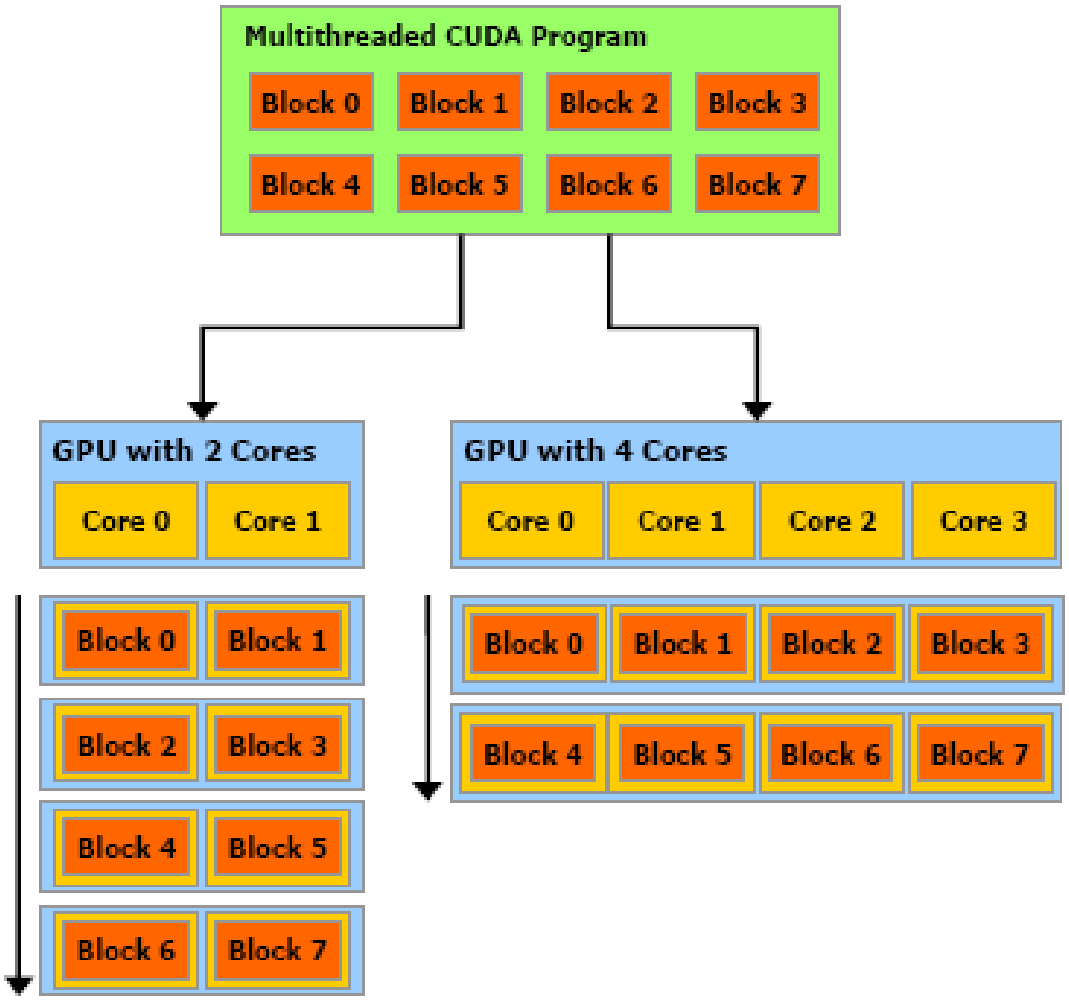


Image From: http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf

Kernel Call Syntax

- Kernels are called with the <<< >>> syntax
- Specifies certain kernel parameters
- <<<Dg, Db, Ns, S>>>
- Where:
  - Dg = dimensions of the grid (type dim3)
  - Db = dimensions of the block (type dim3)
  - Ns = number of bytes shared memory dynamically allocated / block (type size_t). 0 default
  - S = associated cudaStream_t. 0 default

CUDA Function Type Qualifiers

- Kernels are defined as __global__. This specifies that the function runs on the device and is callable from the host only
- __device__ and __host__ are other available qualifers
  - __device__ - executed on device, callable only from device
  - __host__ - default if not specified. Executed on host, callable from host only.

Example CUDA Kernel

- Example syntax:
  - Kernel definition:

```
__global__ void kernel(int* dOut, int a, int b)
{
        dOut[blockDim.x*threadIdx.y + threadIdx.x] = a+b;
}
```

  - Kernel call:

```
kernel<<<1,dim3(2,2)>>>(arr,1,2);
```

CUDA Memory Types

- Access to device memory (slowest, not cached), shared memory (faster) and thread registers (fastest)
- Only device memory is directly-accessible from the host
  - A typical approach is to copy a large dataset to device memory. From there it can be brought into faster shared memory and processed
  - 4 cycles for a shared memory access
  - 400-600 cycles required for device memory access

CUDA Memory Types

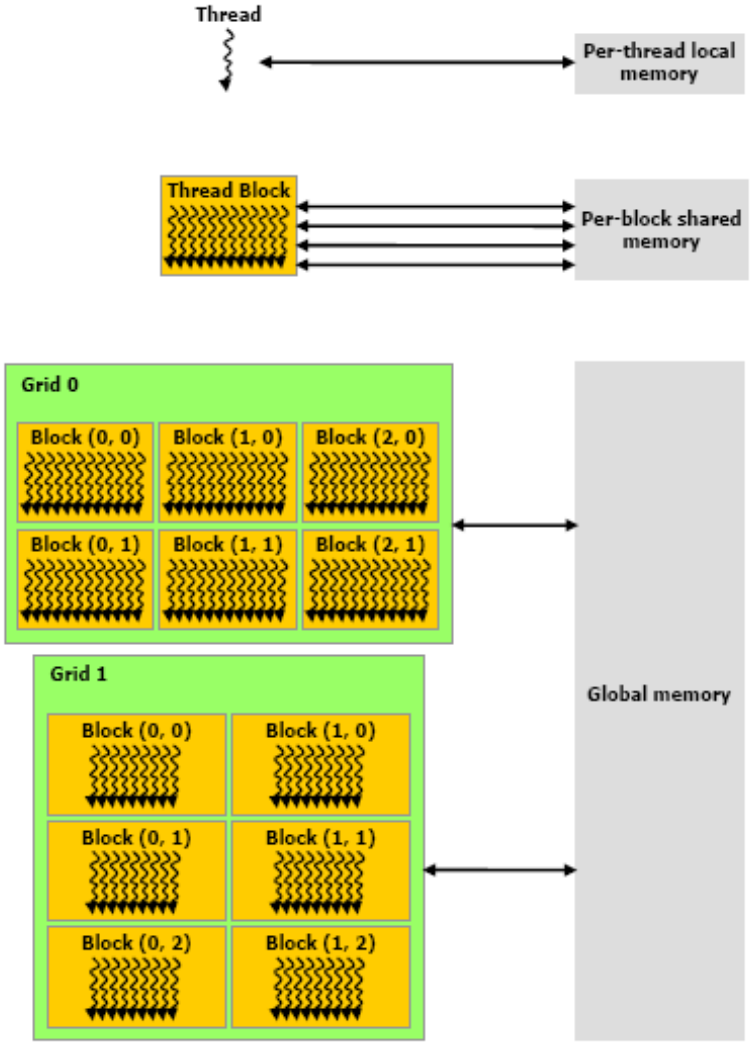
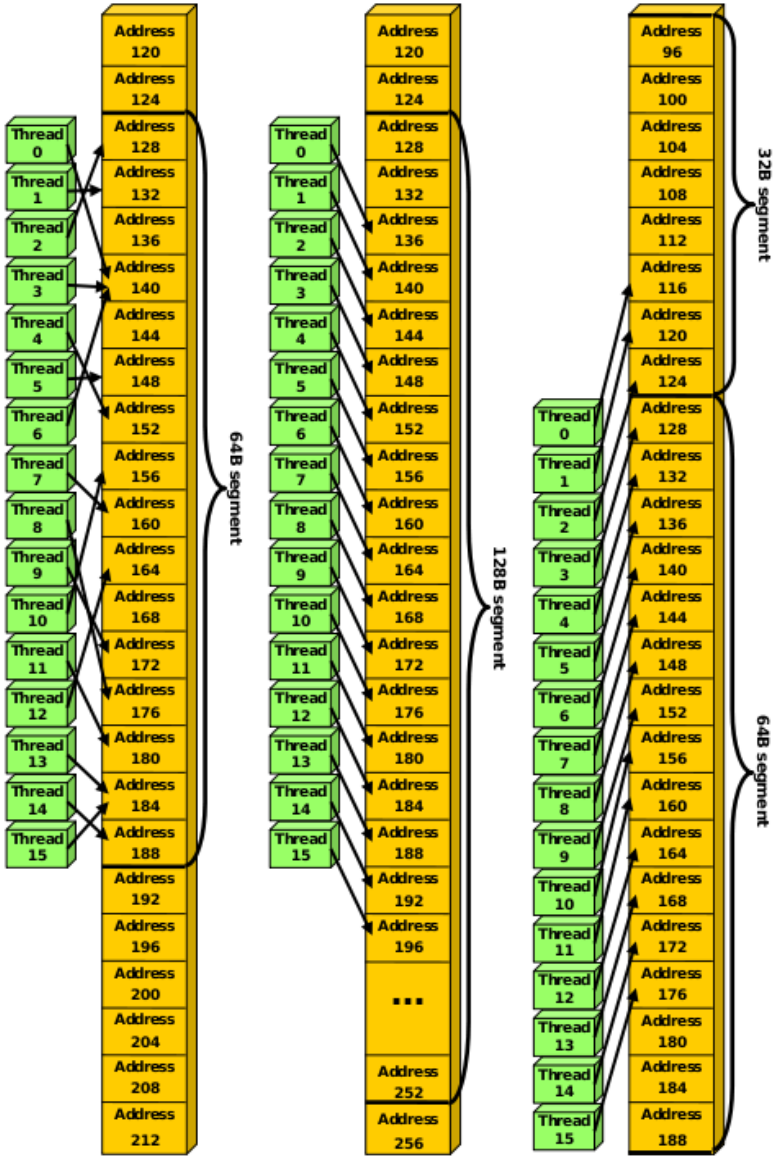


Image From: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf

Memory Access Coalescing

- Coalescing = grouping parallel memory operations into one call
- Compute capability 1.2+ is the least-strict: allows for any type of memory access pattern to be grouped, as long as within a certain segment size
- Transaction coalesced when accessed memory lies in same segment size:
  - 32 bytes if threads access 8-bit words
  - 64 bytes if threads access 16-bit words
  - 128 bytes if threads access 32-bit / 64-bit words



Left: random **float** memory access within a 64B segment, resulting in one memory transaction.
Center: misaligned **float** memory access, resulting in one transaction.
Right: misaligned **float** memory access, resulting in two transactions.

Figure 5-4. Examples of Global Memory Access by Devices with Compute Capability 1.2 and Higher

Image From: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf

Syntax for Using CUDA Device Memory

- cudaError_t cudaMalloc(void** devPtr, size_t size)
  - Allocates size_t bytes of device memory pointed to by devPtr
  - Returns cudaSuccess for no error

- cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, enum cudaMemcpyKind kind)
  - Dst = destination memory address
  - Src = source memory address
  - Count = # bytes to copy
  - Kind = type of transfer
    - cudaMemcpyHostToHost
    - cudaMemcpyHostToDevice
    - cudaMemcpyDeviceToHost
    - cudaMemcpyDeviceToDevice

Syntax for CUDA Device Memory Cont.

- cudaError_t cudaFree(void* devPtr)
  - Frees memory allocated with cudaMalloc
  - Analogous to malloc/free in C

CUDA Shared Memory

- The __shared__ qualifier declares a variable that:
  - Resides in shared memory of a thread block
  - Has lifetime of the block
  - Is only accessible from all threads in the block
- Ex:
  - Declared with “extern __shared__”
    - Size specified dynamically in kernel call (Ns option)
    - Example: extern __shared__ float sArray[];
  - All shared memory uses the same beginning offset
    - Means that “multiple shared arrays” are simulated with a single contiguous array + offsets

Some Extra CUDA Syntax

- #include <cuda_runtime.h>
- cudaSetDevice must be called before executing kernels
- When finished with the device, cudaThreadFinalize should be called

Compiling CUDA Code

- Done with the nvcc compiler
- Nvcc invokes different tools at different stages
- Workflow:
  - Device code is separated from host code
  - Device code compiled into binary (cubin object)
  - Host compiler (gcc/g++) is invoked on host code
  - The two are linked

Compiling CUDA Code Contd.

- On magic, the following flags are needed to compile CUDA code:
  - -I/usr/local/cuda/include
  - -L/usr/local/cuda/lib
  - -lcudart
- A full Makefile will be shown in examples

More Info About CUDA

- Lib, include, bin directories for CUDA are located at /usr/local/cuda on magic
- Magic is running CUDA 2.3
- CUDA 2.3 documentation / examples:
  - http://developer.nvidia.com/object/cuda_2_3_downloads.html
- Especially check out the programming guide:
  - http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf

Submitting a CUDA Job on Magic

- Use queueing system
- Create batch script and submit with qsub command
- Example:

```
#
#PBS -N tspgenetic
#PBS -l walltime=03:00:00
#PBS -l nodes=ci-xeon-2+ci-xeon-3+ci-xeon-4+ci-xeon-5:ppn=1
#PBS -j oe
#
cd $PBS_O_WORKDIR
mpirun -mca btl ^openib,udapl -np 4 -machinefile machinefile
    ./tspgenetic
```

- *Note – past issues with 'ppn' specification line on magic, have had to explicitly declare nodes used in script (with above bolded lines) to request one instance per node*

EXAMPLE: “ADVANCED HELLO WORLD”

(USING CUDA, OPENMP, MPI)

Arbitrary Kernel Example

```
// kernel.cu
//
// An arbitrary kernel
#ifndef _BURN_KERNEL_H_
#define _BURN_KERNEL_H_

extern "C"
{
  __extern__ float shared[];
  __global__ void kernel()
  {
    float a = 3.0 * 5.0;
    float b = (a * 50) / 4;
    int pos = threadIdx.y*blockDim.x+threadIdx.x;
    shared[pos] = b;
  }
}

#endif
```

Example with OpenMPI, OpenMP, CUDA

```
// A simple "hello world" using CUDA, OpenMP, OpenMPI
// Matt Heavner
using namespace std;
#include <stdio.h>
#include <cuda_runtime.h>
#include <stdlib.h>
#include <omp.h>
#include <mpi.h>
#include "kernel.cu“


char processor_name[MPI_MAX_PROCESSOR_NAME];
bool initDevice();
extern "C" void kernel();


bool initDevice()
{
  printf("Init device %d on
      %s\n",omp_get_thread_num(),processor_name);
  return (cudaSetDevice(omp_get_thread_num()) ==
      cudaSuccess);
}
```

```
int main(int argc, char* argv[])
{
  int numprocs,namelen,rank,devCount;
  int val = 0;
  MPI_Status stat;

// Initialize MPI
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Get_processor_name(processor_name, &namelen);
printf("Hello from %d on %s out of
%d\n",(rank+1),processor_name,numprocs);

if (cudaGetDeviceCount(&devCount) != cudaSuccess)
{
  printf("Device error on %s\n!",processor_name);
  MPI_Finalize();
  return 1;
}
// Test MPI message passing
if (rank == 0){
  val= 3;
  for (inti=0; i<numprocs; i++)
  MPI_Send(&val,1,MPI_INT,i,0,MPI_COMM_WORLD);
}
MPI_Recv(&val,1,MPI_INT,0,0,MPI_COMM_WORLD,&stat
);
```

Example with OpenMPI, OpenMP, CUDA cont.

```
if (val== 3)
   cout<< rank << " properly received via MPI!" << endl;
else
   cout<< rank << " had an error receiving over MPI!" << endl;


 // Run one OpenMP thread per device per MPI node
 #pragma omp parallel num_threads(devCount)
 if (initDevice()) {
    // Block and grid dimensions
    dim3 dimBlock(12,12);
    kernel<<<1,dimBlock,dimBlock.x*dimBlock.y*sizeof(float)>>>();
    cudaThreadExit();
 }
 else
  {
     printf("Device error on %s\n",processor_name);
  }
  MPI_Finalize();
  return 0;
}
```

Example Makefile

```
CC=/usr/local/cuda/bin/nvcc
CFLAGS= -I/usr/lib64/openmpi/1.2.7-gcc/include -I/usr/local/cuda/include -
    Xcompiler –fopenmp
LDFLAGS= -L/usr/lib64/openmpi/1.2.7-gcc/lib -L/usr/local/cuda/lib
LIB= -lgomp -lcudart –lmpi
SOURCES= helloworld.cu
EXECNAME= hello

all:
    $(CC) -Xptxas -v --gpu-architecture sm_13  -o $(EXECNAME)
    $(SOURCES) $(LIB) $(LDFLAGS) $(CFLAGS)

clean:
    rm *.o *.linkinfo
```

PAST PROJECT 1: TRAVELING SALESMAN

Genetic Algorithm, Brief Details

- Seek to perform some optimization
- Propose set of candidate solutions
- Evaluate “fitness” of each solution
- Create a new population of candidates that recombines previous candidates
  - Recombination probabilities weighted by fitness
  - Throw in some randomness
  - “Breeding”
- Repeat until termination criteria met

Traveling Salesman Problem, Brief

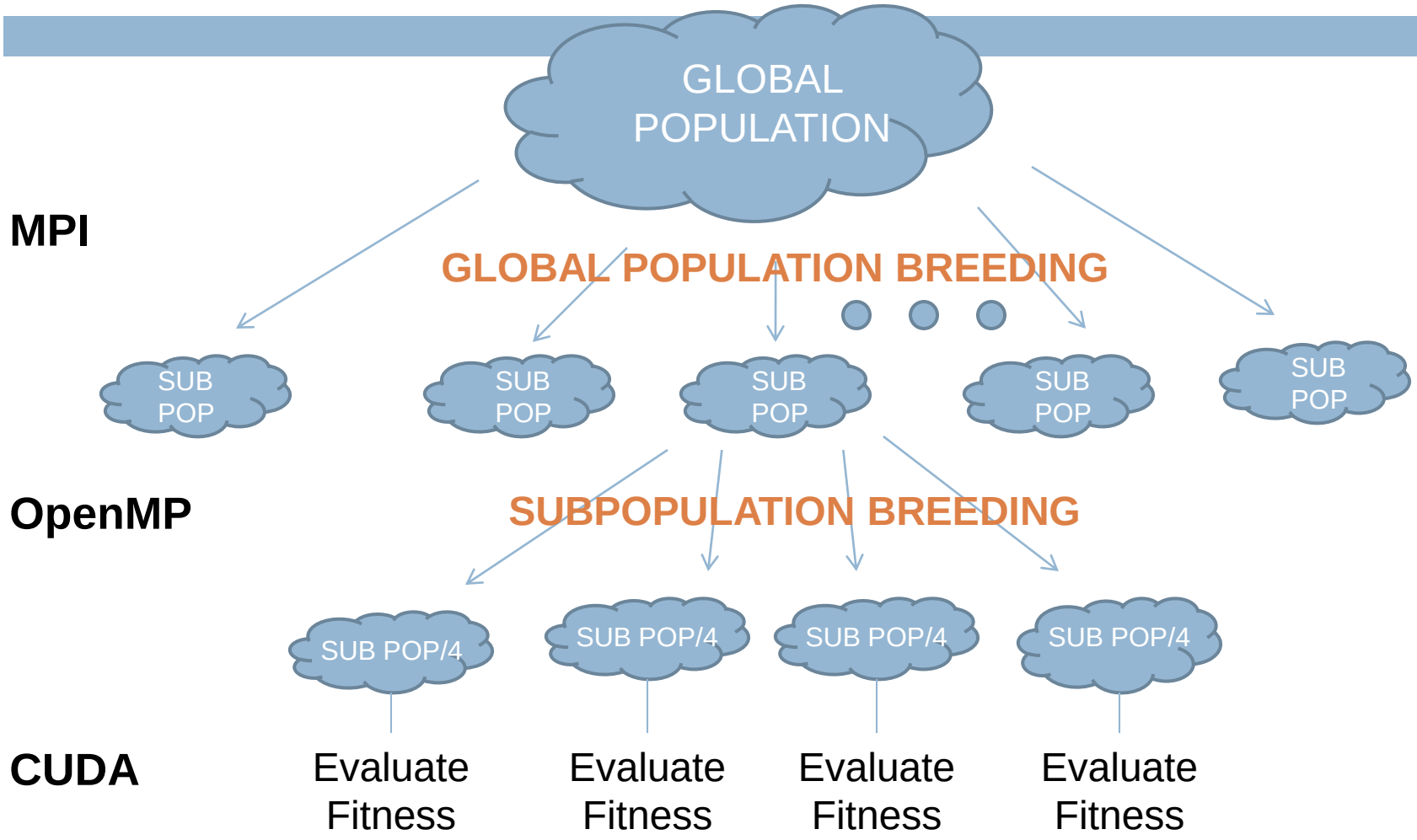
- Given a set of cities, each with a specified coordinate.
- Find a route that visits all cities with the smallest total distance

Mapping to Traveling Salesman

- Given an array of pairs representing city locations
- Candidate = permutation of array indices

- Fitness = 1/(distance of path)
  - We seek to maximize fitness → minimize distance

Levels of Parallelism



Sample Code -- Kernel

```
__global__ void kernel(int* pop, double* fit)
{
    extern __shared__ int sh_pop[];
    double d = 0;
    int pos = blockIdx.x*blockDim.x+threadIdx.x;
    int tIdx = threadIdx.x;

    // Pull this thread's population member into shared memory for fast access
    for (int i=0; i<NUM_CITIES; i++)
       sh_pop[tIdx*NUM_CITIES+i] = pop[pos*NUM_CITIES+i];
    pos = tIdx*NUM_CITIES;
```

Sample Code -- Kernel

```
// Sum distances corresponding to each sequential city pair
    double prevLocX = citylocs[sh_pop[pos]][0];
    double prevLocY = citylocs[sh_pop[pos]][1];
    for (int i=1; i<NUM_CITIES; i++)
    {
       pos = tIdx*NUM_CITIES+i;
       d = d + dist(prevLocX,prevLocY,citylocs[sh_pop[pos]][0],citylocs[sh_pop[pos]][1]);
       prevLocX = citylocs[sh_pop[pos]][0];
       prevLocY = citylocs[sh_pop[pos]][1];
    }
    // Also need distance from last location to first
    pos = tIdx*NUM_CITIES;
    d = d + dist(prevLocX,prevLocY,citylocs[sh_pop[pos]][0],citylocs[sh_pop[pos]][1]);
    fit[blockIdx.x*blockDim.x+threadIdx.x] = d;
}
```

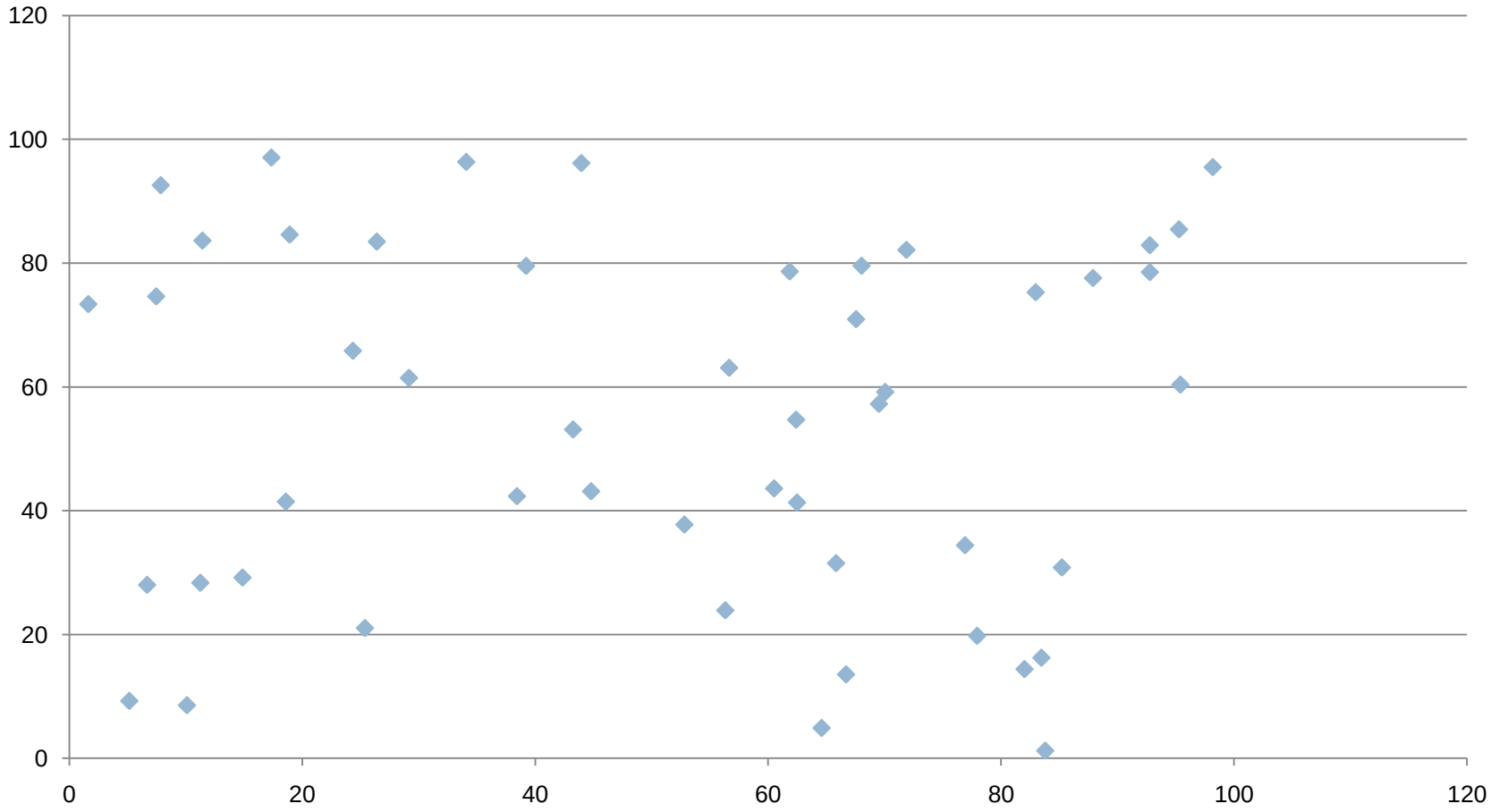
Sample Code

- That's it for the kernel
- Called with:

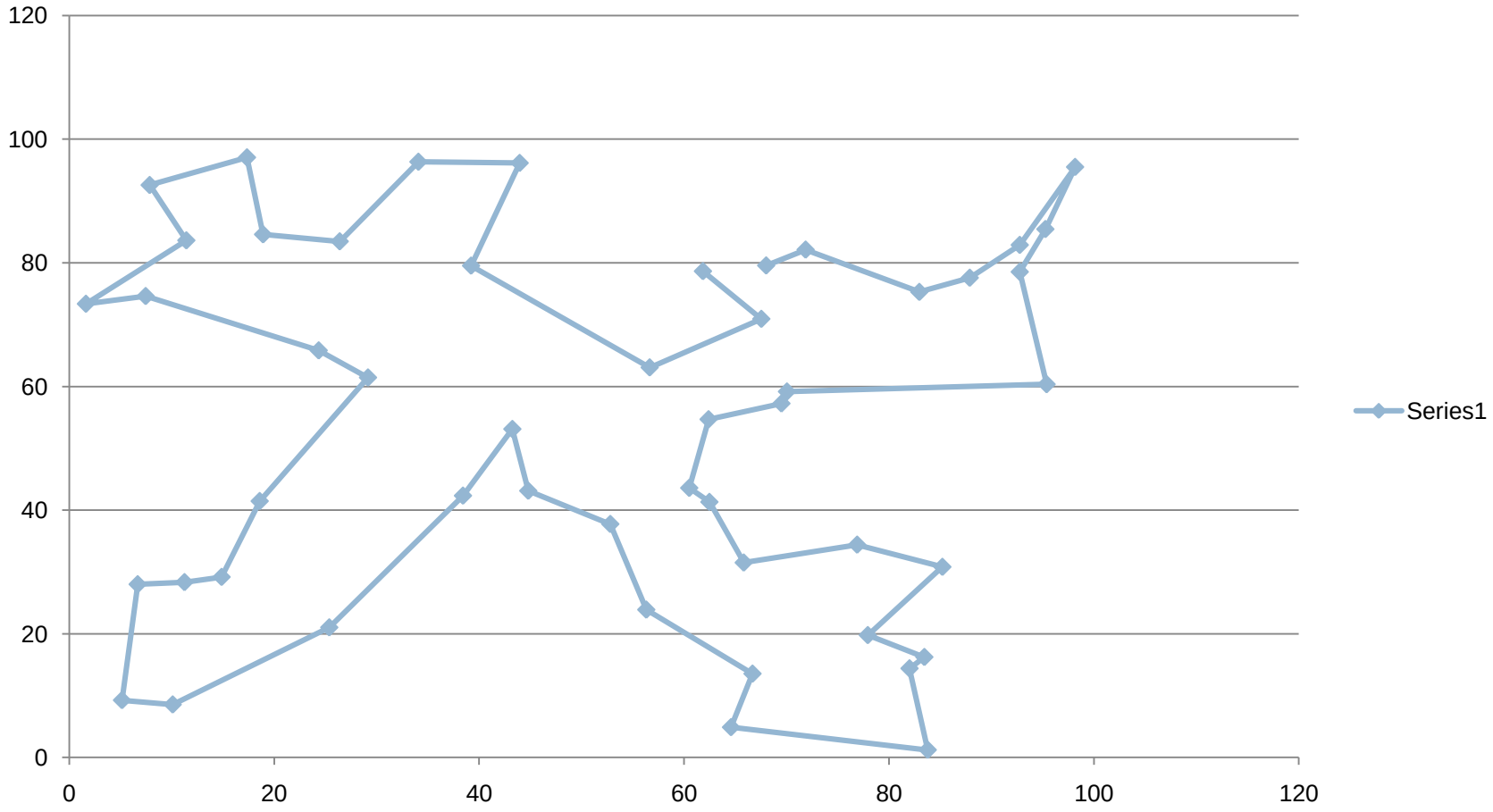
```
kernel<<<dim3(gridSize),dim3(blockSize),blockSize*NUM_CITIES*sizeof(int)>>>(cudapop_dev,cudafit_dev);
```

- Much more code that was run on the host to do "breeding" and send new candidates to the devices
- OpenMP code for "subpopulations", MPI for global population

Results – 50 Cities



Results – 50 Cities



More Information

- Available at:

http://www.cse.buffalo.edu/faculty/miller/Courses/CSE710/710mheavnerTSP.pdf

PAST PROJECT 2: PAGE RANK LEARNING

Dataset / Problem

- Yahoo's Learning to Rank dataset
- Each element in dataset is an online search query
- 700 features in each query, mapped to a corresponding relevance label in {0,1,2,3,4}.
- Seek to find mapping from these features to one of the five labels

Sample Data

1 29922 0 0 0 0 0 0 0 0.90099 0 0 0 0.064688 0.94562 0 0 0 0.047949 0 0 0 0.30267 0 0 0 0.88089 0 0.029732 0 0 0.88748 0 0 0 0.77289 0 0.75339 0.94682 0.81146 0 0 0 0 0.22927 0 0 0 0 0 0 0 0 0 0 0.45297 0 0.82108 0 0 0 0.55283 0 0.74035 0 0 0.74935 0 0 0.88136 0.19638 0 0.98686 0 0 0 0 0 0 0 0 0.28036 0 0 0 0 0 0 0 0 0.22562 0.6088 0 0 0.89646 0 0 0 0.31419 0 0 0 0 0 0 0 0 0 0.76297 0.043092 0 0 0 0.71173 0 0 0 0 0 0.28548 0.53171 0 0 0.74783 0 0 0.5439 0.12013 0 0.048097 0 0 0 0 0 0.57695 0 0 0 0 0.89993 0 0 0 0 0 0.34738 0.41309 0 0.28643 0.26978 0.46102 0 0.48066 0.15745 0 0 0 0 0.13779 0 0 0.46417 0 0.57185 0 0 0 0 0 0 0 0.31814 0.97009 0 0 0 0.048097 0.50225 0.31448 0 0 0 0 0 0 0 0.68175 0 0 0 0 0 0 0 0 0 0 0 0.67459 0 0.76854 0 0 0 0 0 0 0 0 0 0 0.83823 0.98945 0.76694 0.71685 0.82348 0 0.72309 0 0.86758 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0.31394 0 0.54671 0 0 0 0.77402 0.8195 0.77251 0 0 0 0.11799 0.46083 0.96879 0 0 0.43185 0 0 0 0.45073 0 0 0.21316 0.5325 0 0 0 0 0.33699 0.17201 0.52083 0.4555 0 0 0 0 0 0 0 0.2626 0 0.043553 0 0 0 0 0 0 0.61591 0 0 0 0 0 0 0 0 0 0 0 0 0 0.36042 0 0 0.18101 0 0 0.99311 0.84558 0.90743 0 0 0.88089 0 0 0 0 0 0 0 0 0.5152 0 0 0 0 0 0 0 0.33049 0 0 0.89975 0.74933 0.51489 0 0.90367 0 0 0.52273 0 0.46249 0 0 0 0.7597 0 0 0 0.57348 0 0.93733 0.21332 0.95796 0.48687 0.71396 0 0 0 0 0.16026 0 0 0.55205 0.95921 0 0 0 0 0 0.46879 0 0 0 0.1358 0 0 0 0 0.88089 0.55649 0 0 0.69826 0.73644 0.60384 0.072052 0.28912 0.90743 0.18164 0.45376 0 0 0 0 0 0 0 0 0 0.47253 0 0 0.48734 0 0 0 0 0 0 0.67807 0.81338 0 0 0 0.14543 0 0 0 0.10021 0 0 0.14348 0 0.98478 0 0 0 0 0 0.75829 0 0 0 0 0 0 0.0036246 0 0 0 0.048097 0 0 0 0 0.21696 0 0 0.58714 0.031957 0 0 0 0.49986 0.47366 0 0 0 0 0 0 0.70278 0 0.23317 0.46082 0.04203 0.89393 0 0 0 0 0 0 0 0 0 0 0.46705 0 0.59362 0 0 0 0.82223 0.61462 0 0 0.59574 0 0.21868 0 0 0 0 0.19285 0.77139 0 0.77139 0 0 0 0 0.1735 0 0 0 0 0.56657 0.5241 0 0 0 0.95042 0.87328 0 0 0 0 0.22811 0.00040451 0 0 0 0.5702 0.51431 0 0 0 0 0 0 0 0 0 0 0.94138 0 0.07072 0.69319 0 0.42462 0 0.048097 0.25134 0.57705 0.54164 0 0.91876 0 0.096207 0 0 0 0 0 0 0 0 0.45603 0 0 0.88089 0.95921 0 0 0 0 0 0 0 0 0 0.048097 0 0 0 0.33907 0 0 0 0.94551 0.50495 0 0 0.61834 0 0 0.45001 0.93733 0 0 0 0 0 0 0 0 0.50112 0.1163 0 0 0.76446 0 0.20876 0 0 0.46071 0.46047 0.15788 0.048097 0 0 0 0.46147 0 0 0 0 0 0 0 0 0 0 0.5325 0 0 0 0 0.48908 0 0 0.8042 0.51356 0 0 0 0 0 0.030927 0.72248 0 0.86105 0.25782 0.048097 0 0 0 0 0 0 0.92832 0 0 0 0 0 0 0 0 0.79197 0 0 0 0 0 0.48794 0 0 0.89975 0 0 0 0.00040451 0 0.24588 0.74254 0 0 0 0 0 0 0 0 0 0 0 0.23083 0 0 0 0 0 0 0.5043 0 0 0.046567 0 0 0

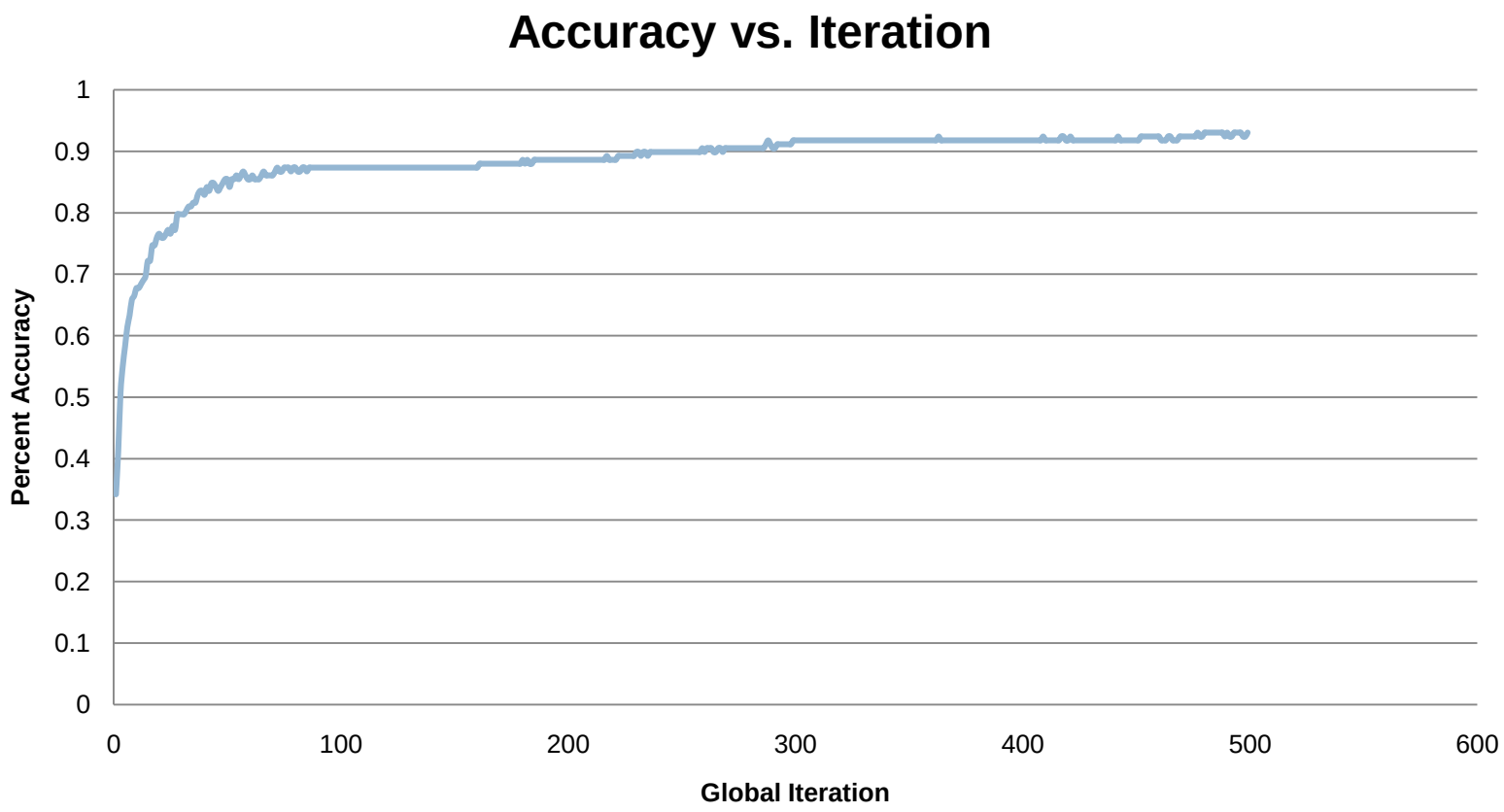
Approach

- Applied genetic algorithm framework developed for TSP to learn a weighting function to map vector to label

$$Label = \left\lfloor \left(\sum_{i=1}^{700} w_i x_i \right) * \frac{1}{Z} \right\rfloor, Z = normalization\ term$$

- So candidates in this problem were a 700-item weight vector
- Fitness was percentage of queries correctly matched in training dataset

Accuracy Results



Simple linear function may have been too general to learn any better

Questions?

- Examples were mostly for the GA framework I worked on, but CUDA is great for anything “data-parallel”
- But you’ll learn the most by reading the programming guide and checking out CUDA SDK examples!
- CUDA 2.3 documentation / examples:
  - http://developer.nvidia.com/object/cuda_2_3_downloads.html
- Especially check out the programming guide:
  - http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf