

Parallel Sorting Algorithms

1 Sorting in C and C++

- using `qsort` in C
- using STL `sort` in C++

2 Bucket Sort for Distributed Memory

- bucket sort in parallel
- communication versus computation

3 Quicksort for Shared Memory

- partitioning numbers
- quicksort with OpenMP
- parallel sort with Intel TBB

MCS 572 Lecture 12
Introduction to Supercomputing
Jan Verschelde, 19 September 2016

Parallel Sorting Algorithms

1 Sorting in C and C++

- using `qsort` in C
- using `STL sort` in C++

2 Bucket Sort for Distributed Memory

- bucket sort in parallel
- communication versus computation

3 Quicksort for Shared Memory

- partitioning numbers
- quicksort with OpenMP
- parallel sort with Intel TBB

using qsort

C provides an implementation of quicksort. The prototype is

```
void qsort ( void *base, size_t count, size_t size,
            int (*compar) (const void *element1,
                           const void *element2) );
```

qsort sorts an array whose first element is pointed to by base and contains count elements, of the given size.

The function compar returns

- **-1 if $\text{element1} < \text{element2}$,**
- **0 if $\text{element1} = \text{element2}$,**
- **+1 if $\text{element1} > \text{element2}$.**

We will apply qsort to sort a random sequence of doubles.

generating and writing numbers

```
void random_numbers ( int n, double a[n] )
{
    int i;
    for(i=0; i<n; i++)
        a[i] = ((double) rand()) / RAND_MAX;
}

void write_numbers ( int n, double a[n] )
{
    int i;
    for(i=0; i<n; i++) printf("%.15e\n", a[i]);
}
```

using qsort

```
int compare ( const void *e1, const void *e2 )
{
    double *i1 = (double*)e1;
    double *i2 = (double*)e2;
    return ((*i1 < *i2) ? -1 : (*i1 > *i2) ? +1 : 0);
}
```

in the function `main()`:

```
double *a;
a = (double*)calloc(n,sizeof(double));
random_numbers(n,a);

qsort((void*)a,(size_t)n,sizeof(double),compare);
```

code to time qsort

We use the command line to enter the dimension and to toggle off the output.

To measure the CPU time for sorting:

```
clock_t tstart, tstop;  
tstart = clock();  
qsort((void*)a, (size_t)n, sizeof(double), compare);  
tstop = clock();  
printf("time elapsed : %.4lf seconds\n",  
      (tstop - tstart) / ((double) CLOCKS_PER_SEC));
```

timing qsort on 3.47GHz Intel Xeon

```
$ time /tmp/time_qsort 1000000 0
time elapsed : 0.2100 seconds
real      0m0.231s
user      0m0.225s
sys       0m0.006s
$ time /tmp/time_qsort 10000000 0
time elapsed : 2.5700 seconds
real      0m2.683s
user      0m2.650s
sys       0m0.033s
$ time /tmp/time_qsort 100000000 0
time elapsed : 29.5600 seconds
real     0m30.641s
user     0m30.409s
sys      0m0.226s
```

Observe: $O(n \log_2(n))$ is almost linear in n .

Parallel Sorting Algorithms

1 Sorting in C and C++

- using `qsort` in C
- using `STL sort` in C++

2 Bucket Sort for Distributed Memory

- bucket sort in parallel
- communication versus computation

3 Quicksort for Shared Memory

- partitioning numbers
- quicksort with OpenMP
- parallel sort with Intel TBB

using the STL container vector

```
#include <vector>
using namespace std;

vector<double> random_vector ( int n );
// returns a vector of n random doubles

vector<double> random_vector ( int n )
{
    vector<double> v;
    for( int i=0; i<n; i++ )
    {
        double r = (double) rand();
        r = r/RAND_MAX;
        v.push_back(r);
    }
    return v;
}
```

writing STL vectors

```
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

void write_vector ( vector<double> v );
// writes the vector v

void write_vector ( vector<double> v )
{
    for(int i=0; i<v.size(); i++)
        cout << scientific
            << setprecision(15)
            << v[i] << endl;
}
```

using the STL sort

```
#include <vector>
#include <algorithm>
using namespace std;

struct less_than // defines "<"
{
    bool operator() (const double& a,
                      const double& b)
    {
        return (a < b);
    }
};
```

in the main program:

```
sort(v.begin(), v.end(), less_than());
```

timing STL sort on 3.47GHz Intel Xeon

```
$ time /tmp/time_stl_sort 1000000 0
time elapsed : 0.36 seconds
real      0m0.376s
user      0m0.371s
sys       0m0.004s
$ time /tmp/time_stl_sort 10000000 0
time elapsed : 4.09 seconds
real      0m4.309s
user      0m4.275s
sys       0m0.033s
$ time /tmp/time_stl_sort 100000000 0
time elapsed : 46.5 seconds
real     0m48.610s
user    0m48.336s
sys     0m0.267s
```

Different distributions may cause timings to fluctuate.

Parallel Sorting Algorithms

1 Sorting in C and C++

- using `qsort` in C
- using STL `sort` in C++

2 Bucket Sort for Distributed Memory

- bucket sort in parallel
- communication versus computation

3 Quicksort for Shared Memory

- partitioning numbers
- quicksort with OpenMP
- parallel sort with Intel TBB

bucket sort

Given are n numbers, suppose all are in $[0, 1]$.

The algorithm using p buckets proceeds in two steps:

- Partition numbers x into p buckets:
 $x \in [i/p, (i + 1)/p[\Rightarrow x \in (i + 1)\text{th bucket.}$
- Sort all p buckets.

The cost to partition the numbers into p buckets is $O(n \log_2(p))$.

Note: radix sort uses most significant bits to partition.

In the best case: every bucket contains n/p numbers.

The cost of Quicksort is $O(n/p \log_2(n/p))$ per bucket.

Sorting p buckets takes $O(n \log_2(n/p))$.

Total cost is $O(n(\log_2(p) + \log_2(n/p)))$.

parallel bucket sort

On p processors, all nodes sort:

- 1 Root node distributes numbers: processor i gets i th bucket.
- 2 Processor i sorts i th bucket.
- 3 Root node collects sorted buckets from processors.

Is it worth it? Recall: serial cost is $n(\log_2(p) + \log_2(n/p))$.

Cost of parallel algorithm:

- $n \log_2(p)$ to place numbers into buckets,
- $n/p \log_2(n/p)$ to sort buckets.

$$\begin{aligned}\text{speedup} &= \frac{n(\log_2(p) + \log_2(n/p))}{n(\log_2(p) + \log_2(n/p)/p)} \\ &= \frac{1+L}{1+L/p} = \frac{1+L}{(p+L)/p} = \frac{p}{p+L}(1+L), \quad L = \frac{\log_2(n/p)}{\log_2(p)}.\end{aligned}$$

comparing to quicksort

$$\begin{aligned}\text{speedup} &= \frac{n \log_2(n)}{n(\log_2(p) + n/p \log_2(n/p))} \\ &= \frac{\log_2(n)}{\log_2(p) + 1/p(\log_2(n) - \log_2(p))} \\ &= \frac{\log_2(n)}{1/p(\log_2(n) + (1 - 1/p)\log_2(p))}\end{aligned}$$

Example: $n = 2^{20}$, $\log_2(n) = 20$, $p = 2^2$, $\log_2(p) = 2$,

$$\begin{aligned}\text{speedup} &= \frac{20}{1/4(20) + (1 - 1/4)2} \\ &= \frac{20}{5 + 3/2} = \frac{40}{13} \approx 3.08.\end{aligned}$$

Parallel Sorting Algorithms

1 Sorting in C and C++

- using `qsort` in C
- using STL `sort` in C++

2 Bucket Sort for Distributed Memory

- bucket sort in parallel
- communication versus computation

3 Quicksort for Shared Memory

- partitioning numbers
- quicksort with OpenMP
- parallel sort with Intel TBB

communication and computation

The scatter of n data elements costs $t_{\text{start up}} + nt_{\text{data}}$, where t_{data} is the cost of sending 1 data element.

For distributing and collecting of all buckets, the total communication time is $2p \left(t_{\text{start up}} + \frac{n}{p} t_{\text{data}} \right)$.

The computation/communication ratio is

$$\frac{(n \log_2(p) + n/p \log_2(n/p))t_{\text{compare}}}{2p \left(t_{\text{start up}} + \frac{n}{p} t_{\text{data}} \right)}$$

where t_{compare} is the cost for one comparison.

the computation/communication ratio

The computation/communication ratio is

$$\frac{(n \log_2(p) + n/p \log_2(n/p))t_{\text{compare}}}{2p \left(t_{\text{start up}} + \frac{n}{p} t_{\text{data}} \right)}$$

where t_{compare} is the cost for one comparison.

We view this ratio for $n \gg p$, for fixed p , so:

$$\frac{n}{p} \log_2 \left(\frac{n}{p} \right) = \frac{n}{p} (\log_2(n) - \log_2(p)) \approx \frac{n}{p} \log_2(n).$$

The ratio then becomes $\frac{n}{p} \log_2(n) t_{\text{compare}} \gg 2nt_{\text{data}}$.

Thus $\log_2(n)$ must be sufficiently high...

Parallel Sorting Algorithms

1 Sorting in C and C++

- using qsort in C
- using STL sort in C++

2 Bucket Sort for Distributed Memory

- bucket sort in parallel
- communication versus computation

3 Quicksort for Shared Memory

- partitioning numbers
- quicksort with OpenMP
- parallel sort with Intel TBB

a recursive algorithm

```
void quicksort ( double *v, int start, int end ) {  
    if(start < end) {  
        int pivot;  
        partition(v,start,end,&pivot);  
        quicksort(v,start,pivot-1);  
        quicksort(v,pivot+1,end);  
    }  
}
```

where `partition` has the prototype:

```
void partition  
    ( double *v, int lower, int upper, int *pivot );  
/* precondition: upper - lower > 0  
 * takes v[lower] as pivot and interchanges elements:  
 * v[i] <= v[pivot] for all i < pivot, and  
 * v[i] > v[pivot] for all i > pivot,  
 * where lower <= pivot <= upper. */
```

a partition function

```
void partition
( double *v, int lower, int upper, int *pivot )
{
    double x = v[lower];
    int up = lower+1;      /* index will go up */
    int down = upper;      /* index will go down */
    while(up < down)
    {
        while((up < down) && (v[up] <= x)) up++;
        while((up < down) && (v[down] > x)) down--;
        if(up == down) break;
        double tmp = v[up];
        v[up] = v[down]; v[down] = tmp;
    }
    if(v[up] > x) up--;
    v[lower] = v[up]; v[up] = x;
    *pivot = up;
}
```

partition and qsort in main()

```
int lower = 0;
int upper = n-1;
int pivot = 0;
if(n > 1) partition(v,lower,upper,&pivot);

if(pivot != 0)
    qsort((void*)v, (size_t)pivot,
           sizeof(double), compare);

if(pivot != n)
    qsort((void*)&v[pivot+1], (size_t)(n-pivot-1),
           sizeof(double), compare);
```

Parallel Sorting Algorithms

1 Sorting in C and C++

- using qsort in C
- using STL sort in C++

2 Bucket Sort for Distributed Memory

- bucket sort in parallel
- communication versus computation

3 Quicksort for Shared Memory

- partitioning numbers
- **quicksort with OpenMP**
- parallel sort with Intel TBB

a parallel region in main()

```
omp_set_num_threads(2);
#pragma omp parallel
{
    if(pivot != 0)
        qsort((void*)v, (size_t)pivot,
               sizeof(double), compare);
    if(pivot != n)
        qsort((void*)&v[pivot+1], (size_t)(n-pivot-1),
               sizeof(double), compare);
}
```

on dual core Mac OS X at 2.26 GHz

```
$ time /tmp/time_qsort 10000000 0  
time elapsed : 4.0575 seconds
```

```
real      0m4.299s  
user      0m4.229s  
sys       0m0.068s
```

```
$ time /tmp/part_qsort_omp 10000000 0  
pivot = 4721964  
-> sorting the first half : 4721964 numbers  
-> sorting the second half : 5278035 numbers
```

```
real      0m3.794s  
user      0m7.117s  
sys       0m0.066s
```

Speed up: $4.299/3.794 = 1.133$, or 13.3% faster with one extra core.

Parallel Sorting Algorithms

1 Sorting in C and C++

- using qsort in C
- using STL sort in C++

2 Bucket Sort for Distributed Memory

- bucket sort in parallel
- communication versus computation

3 Quicksort for Shared Memory

- partitioning numbers
- quicksort with OpenMP
- parallel sort with Intel TBB

using parallel_sort of the Intel TBB

At the top of the program, add the lines

```
#include "tbb/parallel_sort.h"  
  
using namespace tbb;
```

To sort a number of random doubles:

```
int n;  
double *v;  
v = (double*)calloc(n,sizeof(double));  
random_numbers(n,v);  
parallel_sort(v, v+n);
```

an interactive test run

```
$ /tmp/tbb_sort 4 1
4 random numbers :
3.696845319912231e-01
7.545582678888730e-01
6.707372915329120e-01
3.402865237278335e-01
the sorted numbers :
3.402865237278335e-01
3.696845319912231e-01
6.707372915329120e-01
7.545582678888730e-01
$
```

timing parallel runs

```
$ time /tmp/tbb_sort 10000000 0
```

```
real      0m0.479s  
user      0m4.605s  
sys       0m0.168s
```

```
$ time /tmp/tbb_sort 100000000 0
```

```
real      0m4.734s  
user      0m51.063s  
sys       0m0.386s
```

```
$ time /tmp/tbb_sort 1000000000 0
```

```
real      0m47.400s  
user      9m32.713s  
sys       0m2.073s  
$
```

recommended reading

- Edgar Solomonik and Laxmikant V. Kale: **Highly Scalable Parallel Sorting.** In the proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2010.
- Mirko Rahn, Peter Sanders, and Johannes Singler: **Scalable Distributed-Memory External Sorting.** In the proceedings of the 26th IEEE International Conference on Data Engineering (ICDE), pages 685-688, IEEE, 2010.
- Davide Pasetto and Albert Akhriev: **A Comparative Study of Parallel Sort Algorithms.** In SPLASH'11, the proceedings of the ACM international conference companion on object oriented programming systems languages and applications, pages 203-204, ACM 2011.

Summary + Exercises

In the book of Wilkinson and Allen, bucket sort is described in §4.2.1 and chapter 10 is entirely devoted to sorting algorithms.

Exercises:

- ① Consider the fan out scatter and fan in gather operations and investigate how these operations will reduce the communication cost and improve the computation/communication ratio in bucket sort of n numbers on p processors.
- ② Instead of OpenMP, use Pthreads to run Quicksort on two cores.
- ③ Instead of OpenMP, use the Intel Threading Building Blocks to run Quicksort on two cores.

a Massively Parallel Processor: the GPU

1 Introduction to General Purpose GPUs

- five weeks of lectures left
- our equipment: hardware and software

2 Graphics Processors as Parallel Computers

- the performance gap
- CPU and GPU design
- programming models and data parallelism

MCS 572 Lecture 27
Introduction to Supercomputing
Jan Verschelde, 24 October 2016

a Massively Parallel Processor: the GPU

1 Introduction to General Purpose GPUs

- five weeks of lectures left
- our equipment: hardware and software

2 Graphics Processors as Parallel Computers

- the performance gap
- CPU and GPU design
- programming models and data parallelism

general purpose graphics processing units

Thanks to the industrial success of video game development graphics processors became faster than general CPUs.

General Purpose Graphic Processing Units (GPGPUs) are available, capable of double floating point calculations.

Accelerations by a factor of 10 with one GPGPU are not uncommon.

Comparing electric power consumption is advantageous for GPGPUs.

Thanks to the popularity of the PC market, millions of GPUs are available – every PC has a GPU. This is the first time that massively parallel computing is feasible with a mass-market product.

Example: Actual clinical applications on magnetic resonance imaging (MRI) use some combination of PC and special hardware accelerators.

five weeks left in this course

Topics for the five weeks:

- ① architecture, programming models, scalable GPUs
- ② introduction to CUDA and data parallelism
- ③ CUDA thread organization, synchronization
- ④ CUDA memories, reducing memory traffic
- ⑤ coalescing and applications of GPU computing

We follow the book by David B. Kirk and Wen-mei W. Hwu:
Programming Massively Parallel Processors. A Hands-on Approach.
Elsevier 2010; second edition, 2013.

The site gpgpu.org is a good start for many tutorials.

expectations

Expectations?

- ① design of massively parallel algorithms
- ② understanding of architecture and programming
- ③ software libraries to accelerate applications

Key questions:

- ① Which problems may benefit from GPU acceleration?
- ② Rely on existing software or develop own code?
- ③ How to mix MPI, multicore, and GPU?

The textbook authors use the peach metaphor:

- much of the application code will remain sequential;
- GPUs can dramatically improve easy to parallelize code.

a Massively Parallel Processor: the GPU

1 Introduction to General Purpose GPUs

- five weeks of lectures left
- our equipment: hardware and software

2 Graphics Processors as Parallel Computers

- the performance gap
- CPU and GPU design
- programming models and data parallelism

equipment: hardware and software

Microway workstation kepler

- NVIDIA Tesla K20c general purpose graphics processing unit
 - ① number of CUDA cores: 2,496 (13×192)
 - ② frequency of CUDA cores: 706MHz
 - ③ double precision floating point performance: 1.17 Tflops (peak)
 - ④ single precision floating point performance: 3.52 Tflops (peak)
 - ⑤ total global memory: 4800 MBytes
- CUDA programming model with `nvcc` compiler.

Two Intel E5-2670 (2.6GHz 8 cores) CPUs:

- $2.60\text{ GHz} \times 8\text{ flops/cycle} = 20.8\text{ GFlops/core}$;
- $16\text{ core} \times 20.8\text{ GFlops/core} = 332.8\text{ GFlops}$.

$\Rightarrow 1170/332.8 = 3.5$. One K20c is as strong as $3.5 \times 16 = 56.25$ cores.

CUDA stands for Compute Unified Device Architecture, is a general purpose parallel computing architecture introduced by NVIDIA.

kepler versus pascal

NVIDIA Tesla K20 “Kepler” C-class Accelerator

- 2,496 CUDA cores, $2,496 = 13 \text{ SM} \times 192 \text{ cores/SM}$
- 5GB Memory at 208 GB/sec peak bandwidth
- peak performance: 1.17 TFLOPS double precision

NVIDIA Tesla P100 16GB “Pascal” Accelerator

- 3,586 CUDA cores, $3,586 = 56 \text{ SM} \times 64 \text{ cores/SM}$
- 16GB Memory at 720GB/sec peak bandwidth
- peak performance: 5.3 TFLOPS double precision

Programming model: Single Instruction Multiple Data (SIMD).

- Data parallelism: blocks of threads read from memory, execute the same instruction(s), write to memory.
- Massively parallel: need 10,000 threads for full occupancy.

a Massively Parallel Processor: the GPU

1 Introduction to General Purpose GPUs

- five weeks of lectures left
- our equipment: hardware and software

2 Graphics Processors as Parallel Computers

- the performance gap
- CPU and GPU design
- programming models and data parallelism

comparing flops on GPU and CPU

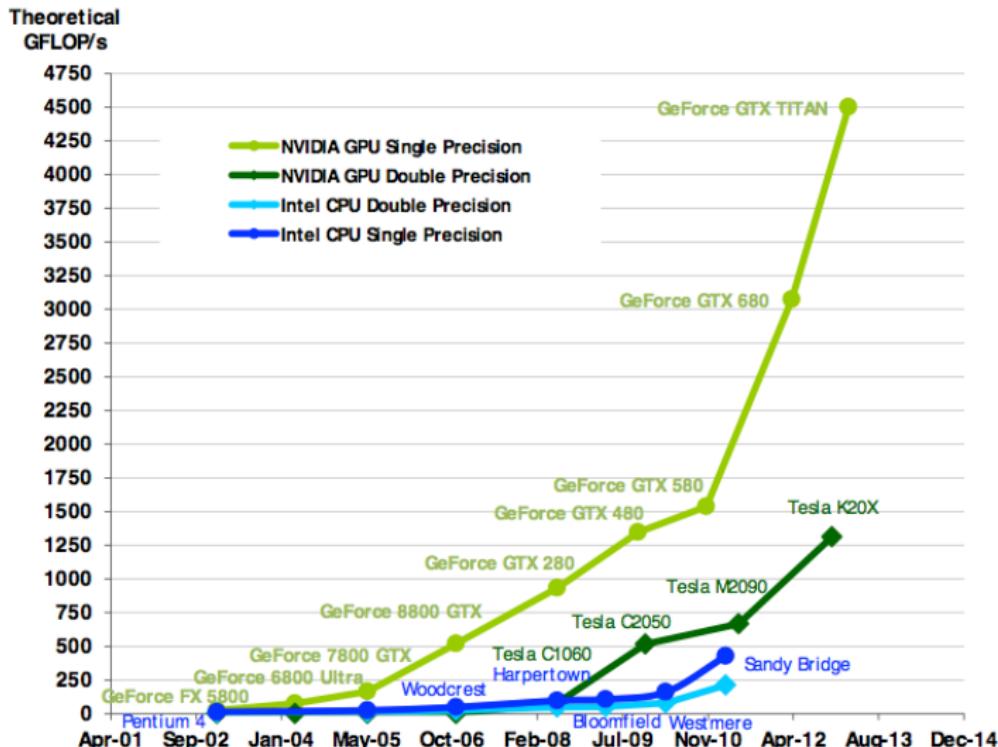
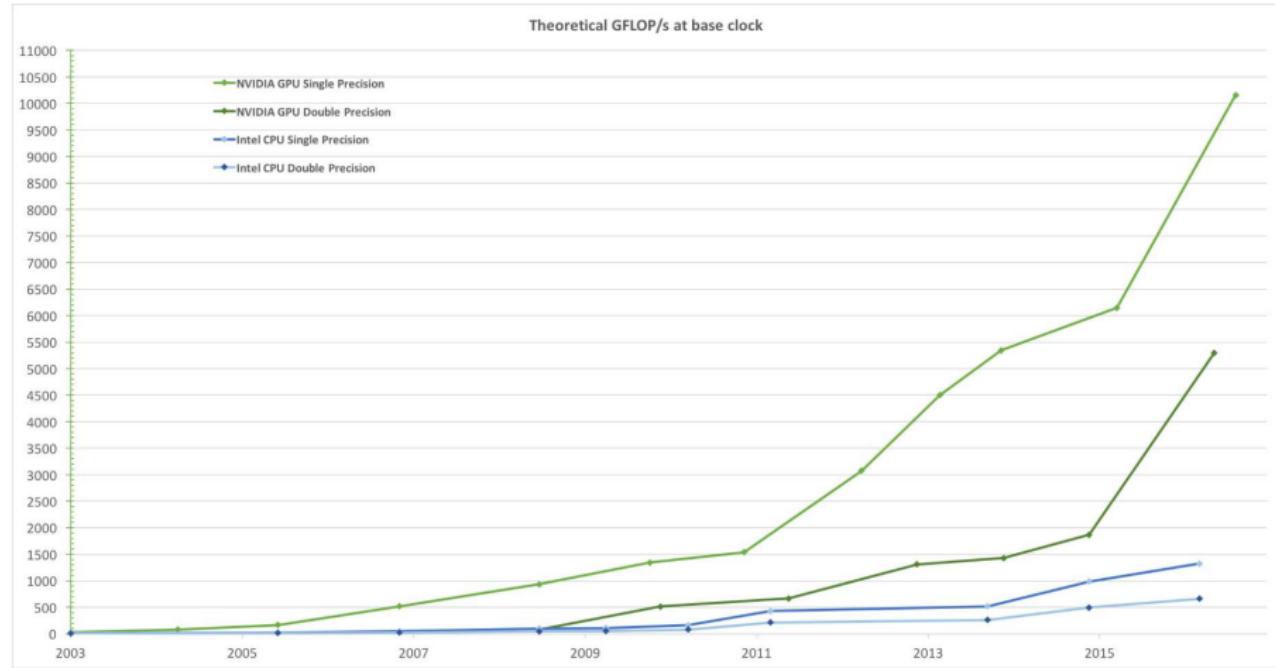


Figure 1 Floating-Point Operations per Second for the CPU and GPU

from the NVIDIA CUDA programming Guide

2016 comparison of flops between CPU and GPU

Figure 1. Floating-Point Operations per Second for the CPU and GPU



comparing memory bandwidths of CPU and GPU

Theoretical GB/s

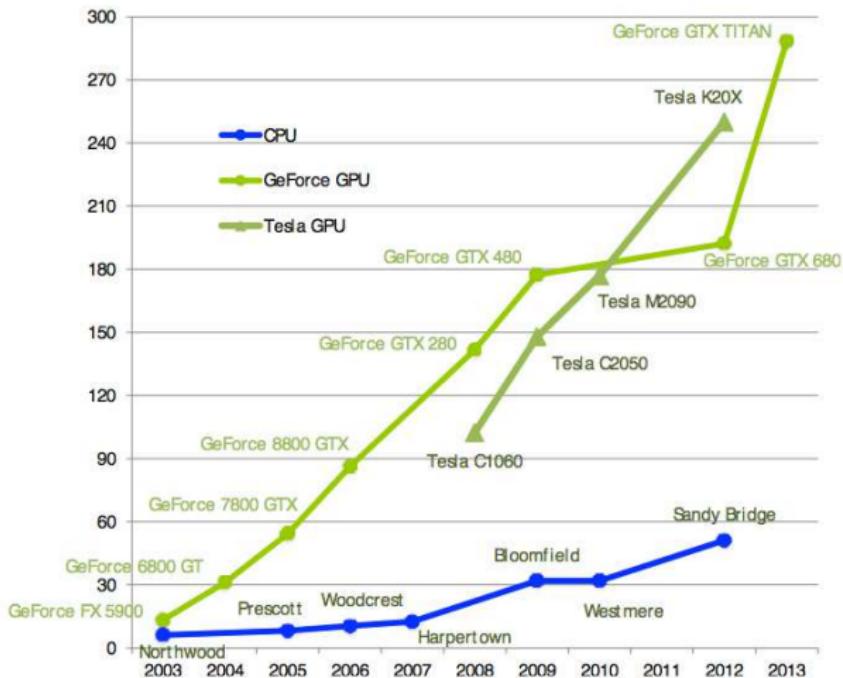
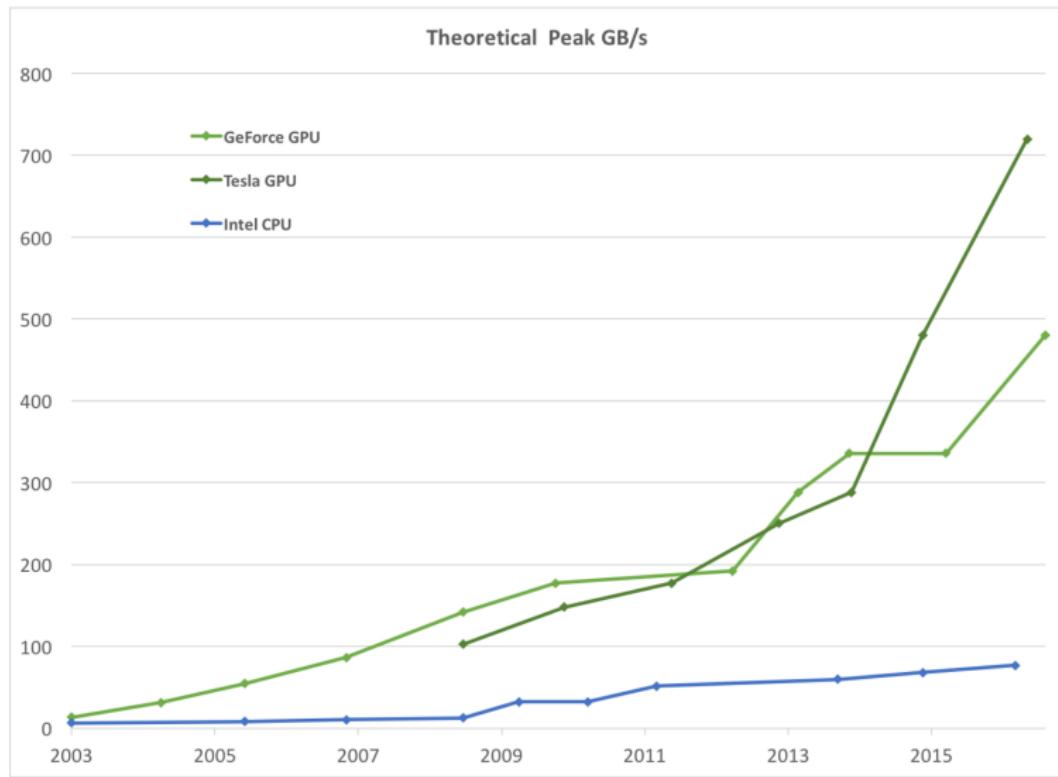


Figure 2 Memory Bandwidth for the CPU and GPU

from the NVIDIA CUDA programming Guide

comparison of bandwidth between CPU and GPU

Figure 2. Memory Bandwidth for the CPU and GPU



memory bandwidth

Graphics chips operate at approximately 10 times the memory bandwidth of CPUs.

Memory bandwidth is the rate at which data can be read from/stored into memory, expressed in bytes per second.

For kepler:

- Intel Xeon E5-2600: the memory bandwidth is 10.66GB/s.
- NVIDIA Tesla K20c: the memory bandwidth is 143GB/s.

For pascal:

- Intel Xeon E5-2699v4: the memory bandwidth is 76.8GB/s.
- NVIDIA Tesla P100: 720GB/s is peak bandwidth.

Straightforward parallel implementations on GPGPUs often achieve directly a speedup of 10, saturating the memory bandwidth.

a Massively Parallel Processor: the GPU

1

Introduction to General Purpose GPUs

- five weeks of lectures left
- our equipment: hardware and software

2

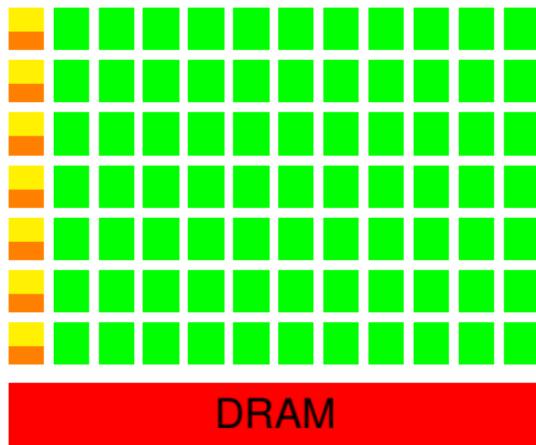
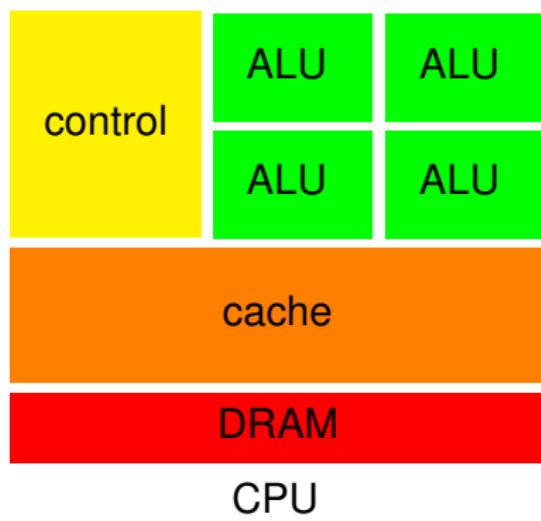
Graphics Processors as Parallel Computers

- the performance gap
- CPU and GPU design
- programming models and data parallelism

CPU and GPU design

CPU: multicore processors have large cores and large caches using control for optimal serial performance.

GPU: optimizing execution throughput of massive number of threads with small caches and minimized control units.



architecture of a modern GPU

- A CUDA-capable GPU is organized into an array of highly threaded Streaming Multiprocessors (SMs).
- Each SM has a number of Streaming Processors (SPs) that share control logic and an instruction cache.
- Global memory of a GPU consists of multiple gigabytes of Graphic Double Data Rate (GDDR) DRAM.
- Higher bandwidth makes up for longer latency.
- The growing size of global memory allows to keep data longer in global memory, with only occasional transfers to the CPU.
- A good application runs 10,000 threads simultaneously.

GeForce 8800 (2007)

16 highly threaded SM's, >128 FPU's, 367 GFLOPS, 768 MB DRAM, 86.4 GB/S Mem BW, 4GB/S BW to CPU



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009 | Memory
ECE 498AL Spring 2010, University of Illinois, Urbana-Champaign

our NVIDIA Tesla K20C and P100 GPUs

The K20C GPU has

- 13 streaming multiprocessors (SM),
- each SM has 192 streaming processors (SP),
- $13 \times 192 = 2496$ cores.

The P100 GPU has

- 56 streaming multiprocessors (SM),
- each SM has 64 streaming processors (SP),
- $56 \times 64 = 3,586$ cores.

Streaming multiprocessors support up to 2,048 threads.

The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*.

Unlike CPU cores, threads are executed in order and there is no branch prediction, although instructions are pipelined.

a Massively Parallel Processor: the GPU

1 Introduction to General Purpose GPUs

- five weeks of lectures left
- our equipment: hardware and software

2 Graphics Processors as Parallel Computers

- the performance gap
- CPU and GPU design
- programming models and data parallelism

programming models

According to David Kirk and Wen-mei Hwu (page 14):

“Developers who are experienced with MPI and OpenMP will find CUDA easy to learn.”

CUDA (Compute Unified Device Architecture) is a programming model that focuses on data parallelism.

On kepler, adjust your .bashrc (the \ means *ignore newline*):

```
export LD_LIBRARY_PATH\  
=$LD_LIBRARY_PATH:/usr/local/cuda-6.5/lib64
```

```
PATH=/usr/local/cuda-6.5/bin:$PATH; export PATH
```

data parallelism

Data parallelism involves

- ① huge amounts of data on which
- ② the arithmetical operations are applied in parallel.

With MPI we applied the SPMD (Single Program Multiple Data) model.

With GPGPU, the architecture is

SIMT = Single Instruction Multiple Thread

An example with large amount of data parallelism is matrix-matrix multiplication in large dimensions.

Available Software Development Tools (SDK), e.g.: BLAS, FFT
from http://www.nvidia.com/object/tesla_software.html

Alternatives and Extensions

Alternatives to CUDA:

- OpenCL (chapter 14) for heterogeneous computing;
- OpenACC (chapter 15) uses directives like OpenMP;
- C++ Accelerated Massive Parallelism (chapter 18).

Extensions:

- Thrust: productivity-oriented library for CUDA (chapter 16);
- CUDA FORTRAN (chapter 17);
- MPI/CUDA (chapter 19).

suggested reading

We covered chapter 1 of the book by Kirk and Hwu.

- NVIDIA CUDA Programming Guide.
Available at developer.nvidia.com
- Victor W. Lee et al: **Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU.**
In *Proceedings of the 37th annual International Symposium on Computer Architecture (ISCA'10)*, ACM 2010.
- W.W. Hwu (editor). **GPU Computing Gems: Emerald Edition.**
Morgan Kaufmann, 2011.

Exercise: visit gpgpu.org.

Notes and audio of a ECE 498 at UIUC, Spring 2009, are at

<https://nanohub.org/resources/7225/supportingdocs>.

Evolution of Graphics Pipelines

1 Understanding the Graphics Heritage

- the era of fixed-function graphics pipelines
- the stages to render triangles

2 Programmable Real-Time Graphics

- programmable vertex and fragment processors
- an example of a programmable pipeline
- unified graphics and computing processors
- GPU computing

MCS 572 Lecture 28
Introduction to Supercomputing
Jan Verschelde, 26 October 2016

Evolution of Graphics Pipelines

1 Understanding the Graphics Heritage

- the era of fixed-function graphics pipelines
- the stages to render triangles

2 Programmable Real-Time Graphics

- programmable vertex and fragment processors
- an example of a programmable pipeline
- unified graphics and computing processors
- GPU computing

evolution of graphics pipelines

Graphics processing units (GPUs) are massively parallel numeric computing processors, programmed in C with extensions.

Understanding the graphics heritage illuminates the strengths and weaknesses of GPUs with respect to major computational patterns.

The history clarifies the rationale behind major architectural design decisions of modern programmable GPUs:

- massive multithreading,
- relatively small cache memories compared to caches of CPUs,
- bandwidth-centric memory interface design.

Insights in the history provide the context for the future evolution.

advanced graphics hardware performance

Three dimensional (3D) graphics pipeline hardware evolved from large expensive systems of the early 1980s to small workstations and then PC accelerators in the mid to late 1990s.

During this period, the performance increased:

- from 50 millions pixels to 1 billion pixels per second,
- from 100,000 vertices to 10 million vertices per second.

This advancement was driven by market demand for high quality, real time graphics in computer applications.

The architecture evolved

- from a simple pipeline for drawing wire frame diagrams
- to a parallel design of several deep parallel pipelines capable of rendering the complex interactive imagery of 3D scenes.

In the mean time, graphics processors became programmable.

Evolution of Graphics Pipelines

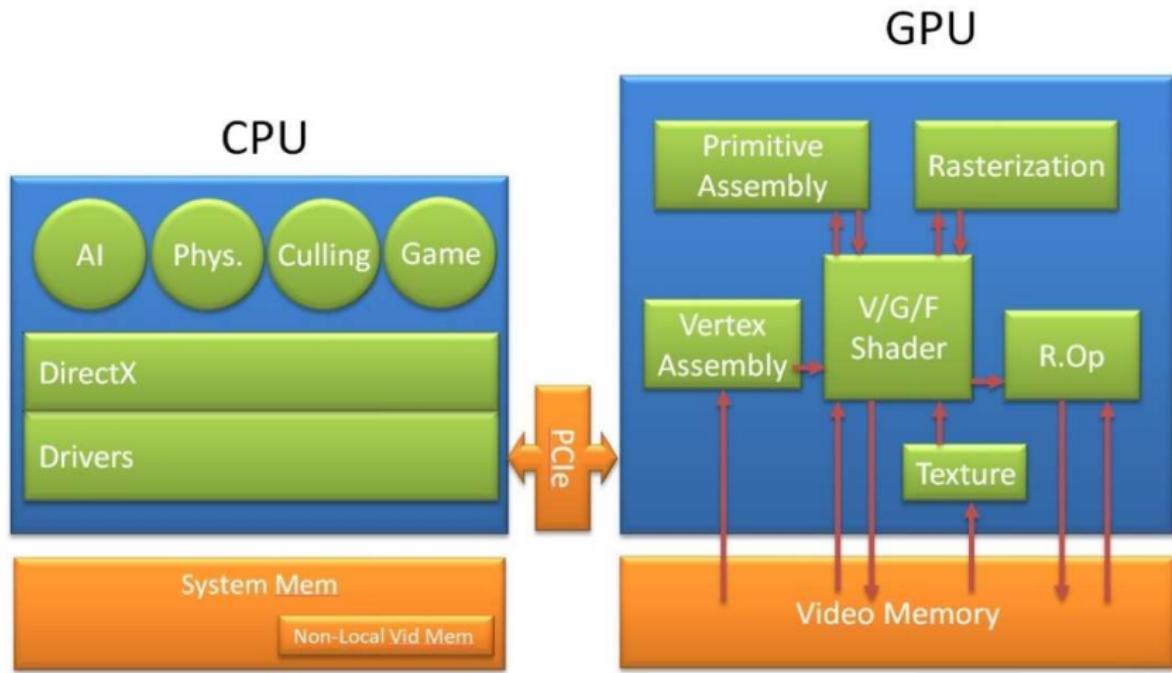
1 Understanding the Graphics Heritage

- the era of fixed-function graphics pipelines
- the stages to render triangles

2 Programmable Real-Time Graphics

- programmable vertex and fragment processors
- an example of a programmable pipeline
- unified graphics and computing processors
- GPU computing

displaying images



from the GeForce 8 and 9 Series GPU Programming Guide (NVIDIA)

rendering triangles

The surface of an object is drawn as a collection of triangles.

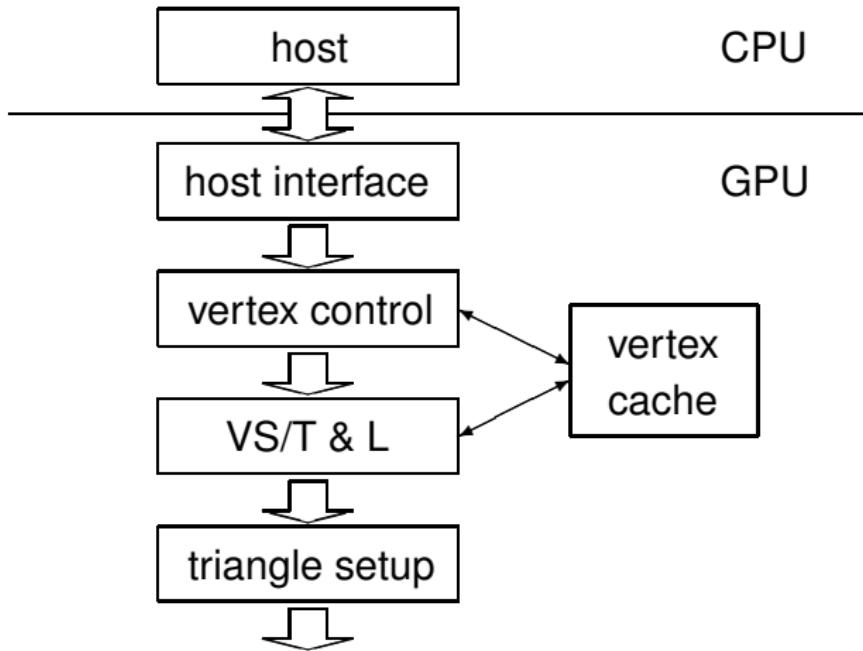
The Application Programming Interface (API) is a standardized layer of software that allows an application (e.g.: a game) to send commands to a graphics processing unit to draw objects on a display.

Examples of such APIs are DirectX and OpenGL.

The host interface (the interface to the GPU)

- receives graphics commands and data from the CPU,
- communicates back the status and result data of the execution.

a fixed-function pipeline – part one



VS/T & L = vertex shading, transform, and lighting

stages in the first part of the pipeline

① vertex control

This stage receives parametrized triangle data from the CPU.
The data gets converted and placed into the vertex cache.

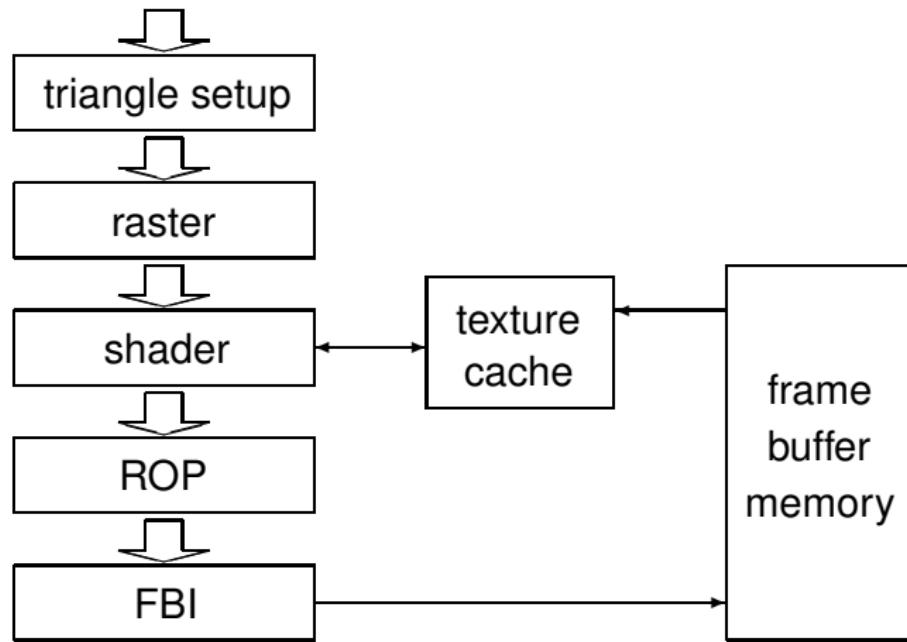
② VS/T & L (vertex shading, transform, and lighting)

The VS/T & L stage transforms vertices and assigns per-vertex values, e.g.: colors, normals, texture coordinates, tangents.
The vertex shader can assign a color to each vertex,
but color is not applied to triangle pixels until later.

③ triangle setup

Edge equations are used to interpolate colors and other per-vertex data across the pixels touched by the triangle.

a fixed-function pipeline – part two



ROP = Raster Operation, FBI = Frame Buffer Interface

the stages in the second part of the pipeline

④ raster

The raster determines which pixels are contained in each triangle. Per-vertex values necessary for shading are interpolated.

⑤ shader

The shader determines the final color of each pixel as a combined effect of interpolation of vertex colors, texture mapping, per-pixel lighting, reflections, etc.

⑥ ROP (Raster Operation)

The final raster operations blend the color of overlapping/adjacent objects for transparency and antialiasing effects.

For a given viewpoint, visible objects are determined and occluded pixels (blocked from view by other objects) are discarded.

finally, the Frame Buffer Interface

7 FBI (Frame Buffer Interface)

The FBI stages manages memory reads from and writes to the display frame buffer memory.

For high-resolution displays, there is a very high bandwidth requirement in accessing the frame buffer.

High bandwidth is achieved by two strategies:

- 1 using special memory designs,
- 2 managing simultaneously multiple memory channels that connect to multiple memory banks.

Evolution of Graphics Pipelines

1 Understanding the Graphics Heritage

- the era of fixed-function graphics pipelines
- the stages to render triangles

2 Programmable Real-Time Graphics

- **programmable vertex and fragment processors**
- an example of a programmable pipeline
- unified graphics and computing processors
- GPU computing

data independence

Stages in graphics pipelines do many floating-point operations on completely independent data, e.g.:

- transforming the positions of triangle vertices,
- generating pixel colors.

This *data independence* as the dominating characteristic is the key difference between the design assumption for GPUs and CPUs.

A single frame, rendered in $1/60^{\text{th}}$ of a second,
might have a million triangles and 6 million pixels.

the vertex shader

Vertex shader programs map the positions of triangle vertices onto the screen, altering their position, color, or orientation.

A vertex shader thread reads a vertex position (x, y, z, w) and computes its position on screen.

Geometry shader programs operate on primitives defined by multiple vertices, changing them or generating additional primitives.

Vertex shader programs and geometry shader programs execute on the vertex shader (VS/T & L) stage of the graphics pipeline.

shader programs

A shader program calculates the floating-point red, green, blue, alpha (RGBA) color contribution to the rendered image at its pixel sample image position.

The programmable vertex processor

- executes programs designated to the vertex shader stage.

The programmable fragment processor

- executes programs designated to the (pixel) shader stage.

For all graphics shader programs, instances can be run in parallel, because each works on independent data, produces independent results, and has no side effects.

This property has motivated the design of the programmable pipeline stages into massively parallel processors.

Evolution of Graphics Pipelines

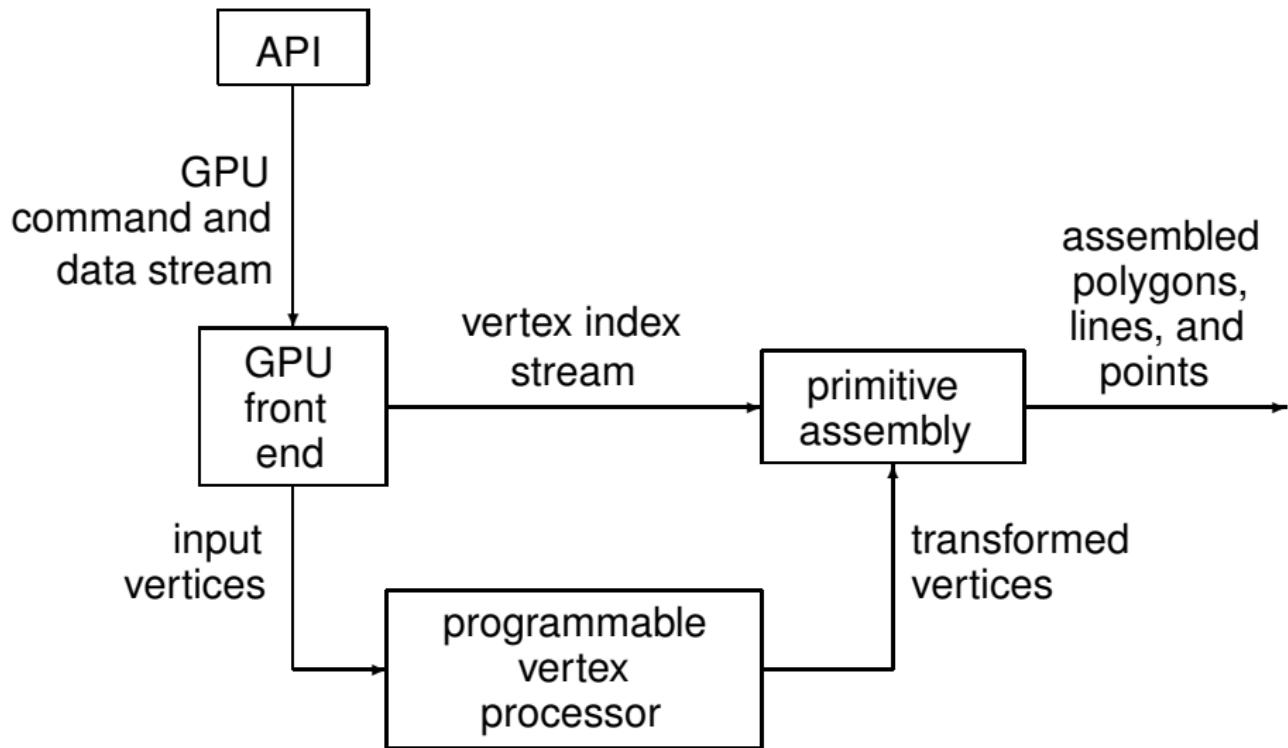
1 Understanding the Graphics Heritage

- the era of fixed-function graphics pipelines
- the stages to render triangles

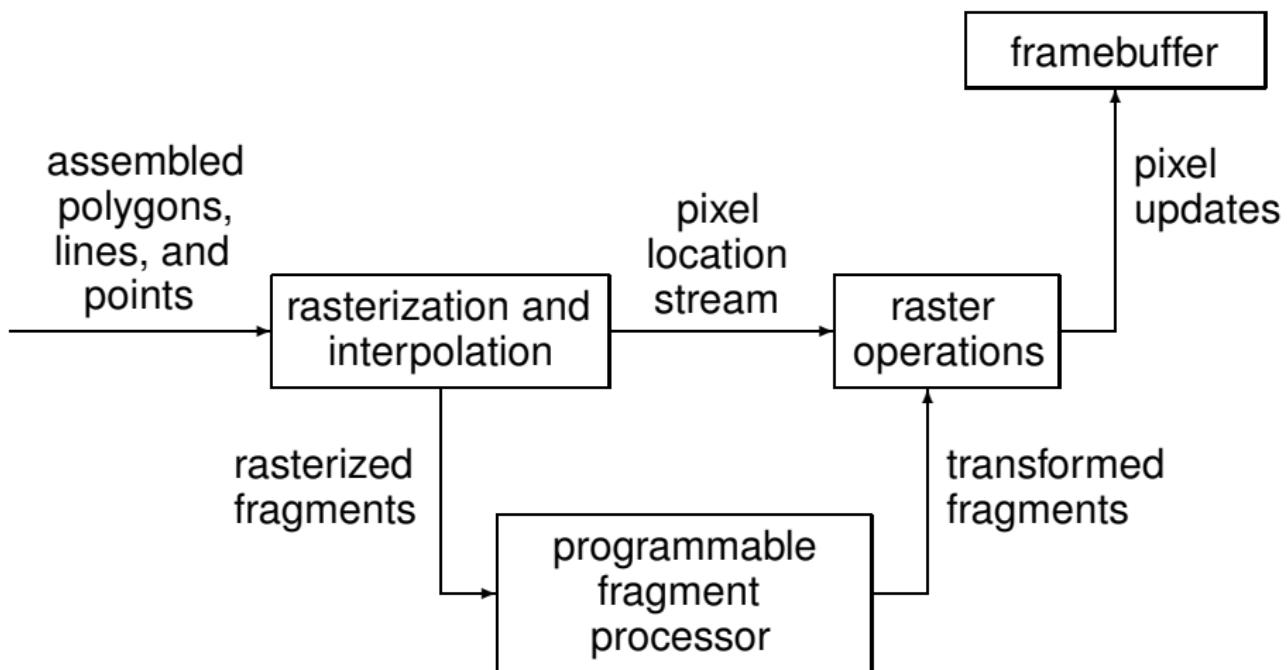
2 Programmable Real-Time Graphics

- programmable vertex and fragment processors
- **an example of a programmable pipeline**
- unified graphics and computing processors
- GPU computing

vertex processor in a pipeline



fragment processor in a pipeline



fixed-function tasks and programmable processors

Between the programmable graphics pipeline stages are dozens of fixed-function stages that perform well-defined tasks far more efficiently than a programmable processor could and which would benefit far less from programmability.

For example, between the vertex processing stage and the pixel (fragment) processing stage is a *rasterizer*.

The rasterizer — it does rasterization and interpolation — is a complex state machine that determines exactly which pixels (and portions thereof) lie within each geometric primitive's boundaries.

The mix of programmable and fixed-function stages is engineered to balance performance with user control over the rendering algorithm.

Evolution of Graphics Pipelines

1 Understanding the Graphics Heritage

- the era of fixed-function graphics pipelines
- the stages to render triangles

2 Programmable Real-Time Graphics

- programmable vertex and fragment processors
- an example of a programmable pipeline
- unified graphics and computing processors**
- GPU computing

unified graphics and computing processors

Introduced in 2006, the GeForce 8800 GPU mapped the separate programmable graphics stages to an array of unified processors.

The graphics pipeline is physically a recirculating path that visits the processors three times, with much fixed-function tasks in between.

More sophisticated shading algorithms motivated a sharp increase in the available shader operation rate, in floating-point operations.

High-clock-speed design made programmable GPU processor array ready for general numeric computing.

Original GPGPU programming used APIs (DirectX or OpenGL): to a GPU everything is a pixel.

GeForce 8800 GPU for GPGPU programming

G80 – Graphics Mode

- The future of GPUs is programmable processing
- So – build the architecture around the processor



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009

ECE 498AL Spring 2010, University of Illinois, Urbana-Champaign

GeForce 8800 GPU with new interface

G80 CUDA mode – A Device Example

- Processors execute computing threads
- New operating mode/HW interface for computing



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE 498AL Spring 2010, University of Illinois, Urbana-Champaign

Evolution of Graphics Pipelines

1 Understanding the Graphics Heritage

- the era of fixed-function graphics pipelines
- the stages to render triangles

2 Programmable Real-Time Graphics

- programmable vertex and fragment processors
- an example of a programmable pipeline
- unified graphics and computing processors
- **GPU computing**

GPU computing

Drawbacks of the GPGPU model:

- ① The programmer must know APIs and GPU architecture well.
- ② Programs expressed in terms of vertex coordinates, textures, shader programs, add to the complexity.
- ③ Random reads and writes to memory are not supported.
- ④ No double precision is limiting for scientific applications.

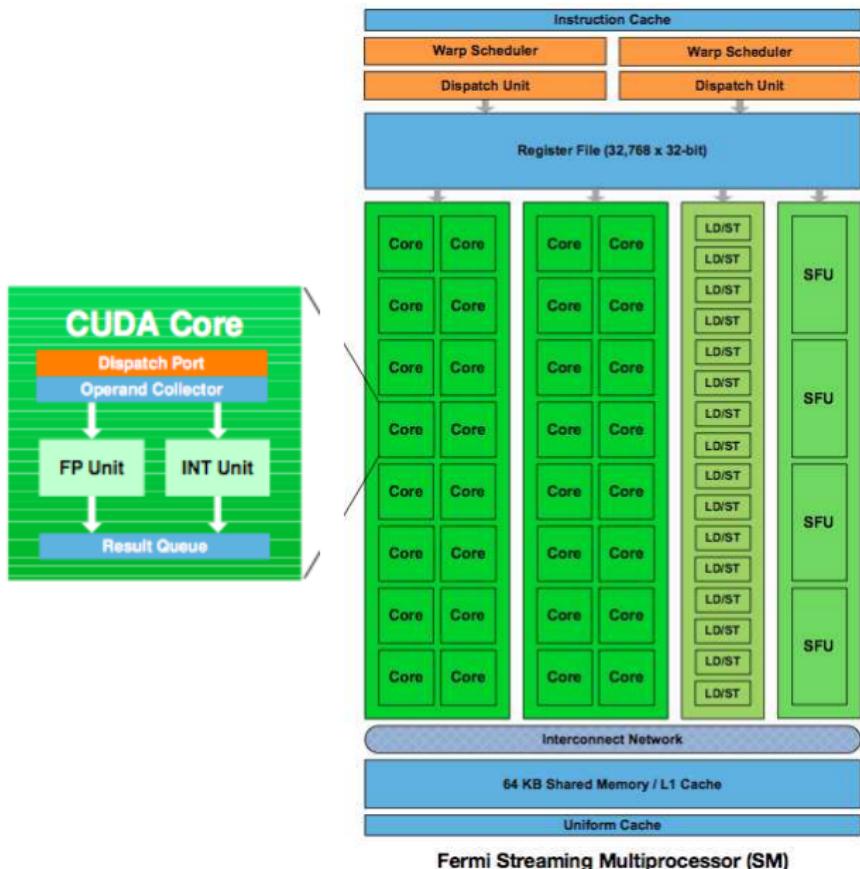
Programming GPUs with CUDA (C extension): GPU computing.

Chapter 2 in the textbook ends mentioning the GT200.

The next generation is code-named Fermi:

- 32 CUDA cores per streaming multiprocessor,
- $8\times$ peak double precision floating point performance over GT200,
- true cache hierarchy, more shared memory,
- faster context switching, faster atomic operations.

the 3rd generation Fermi Architecture

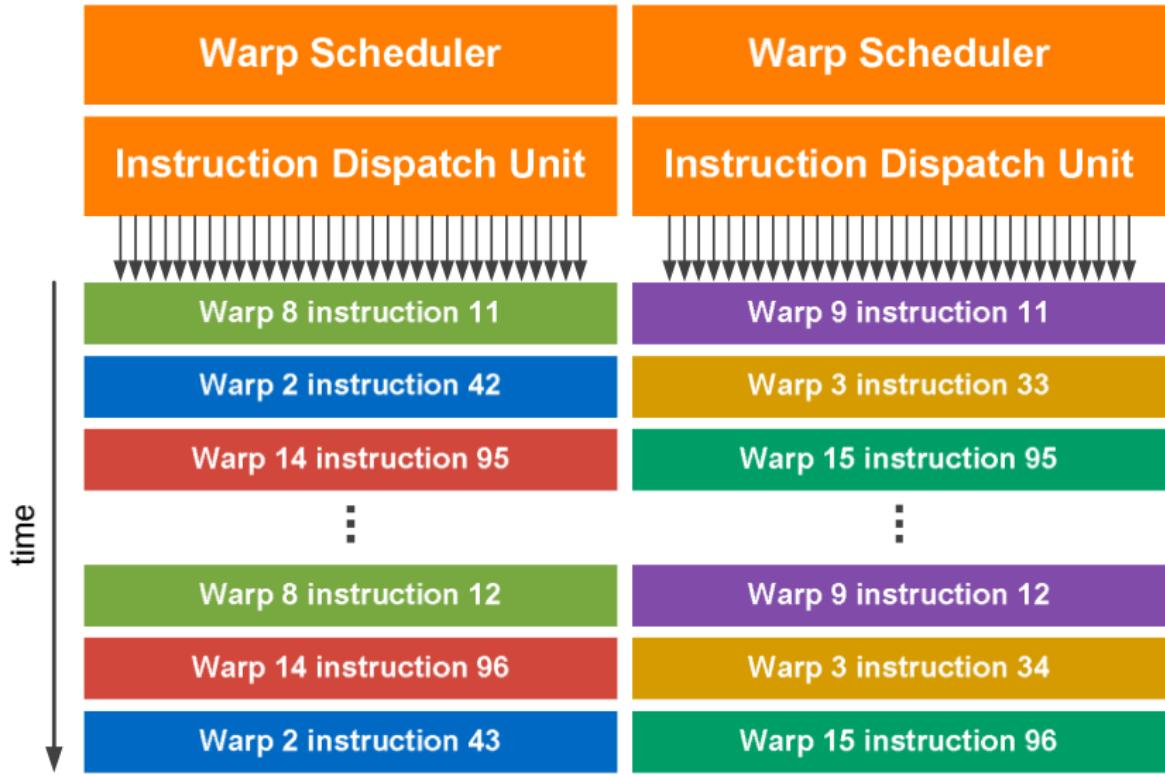


Pascal's Streaming Multiprocessor



from the NVIDIA Tesla P100 Whitepaper

dual warp scheduler



summary table

| GPU | G80 | GT200 | Fermi |
|---|-------------------|---------------------|-----------------------------|
| Transistors | 681 million | 1.4 billion | 3.0 billion |
| CUDA Cores | 128 | 240 | 512 |
| Double Precision Floating Point Capability | None | 30 FMA ops / clock | 256 FMA ops /clock |
| Single Precision Floating Point Capability | 128 MAD ops/clock | 240 MAD ops / clock | 512 FMA ops /clock |
| Special Function Units (SFUs) / SM | 2 | 2 | 4 |
| Warp schedulers (per SM) | 1 | 1 | 2 |
| Shared Memory (per SM) | 16 KB | 16 KB | Configurable 48 KB or 16 KB |
| L1 Cache (per SM) | None | None | Configurable 16 KB or 48 KB |
| L2 Cache | None | None | 768 KB |
| ECC Memory Support | No | No | Yes |
| Concurrent Kernels | No | No | Up to 16 |
| Load/Store Address Width | 32-bit | 32-bit | 64-bit |

the Kepler architecture

| | FERMI GF100 | FERMI GF104 | KEPLER GK104 | KEPLER GK110 |
|--|----------------|----------------|-----------------|-----------------|
| Compute Capability | 2.0 | 2.1 | 3.0 | 3.5 |
| Threads / Warp | 32 | 32 | 32 | 32 |
| Max Warps / Multiprocessor | 48 | 48 | 64 | 64 |
| Max Threads / Multiprocessor | 1536 | 1536 | 2048 | 2048 |
| Max Thread Blocks / Multiprocessor | 8 | 8 | 16 | 16 |
| 32-bit Registers / Multiprocessor | 32768 | 32768 | 65536 | 65536 |
| Max Registers / Thread | 63 | 63 | 63 | 255 |
| Max Threads / Thread Block | 1024 | 1024 | 1024 | 1024 |
| Shared Memory Size Configurations (bytes) | 16K | 16K | 16K | 16K |
| | 48K | 48K | 32K | 32K |
| | | | 48K | 48K |
| Max X Grid Dimension | $2^{16}-1$ | $2^{16}-1$ | $2^{32}-1$ | $2^{32}-1$ |
| Hyper-Q | No | No | No | Yes |
| Dynamic Parallelism | No | No | No | Yes |

Compute Capability of Fermi and Kepler GPUs

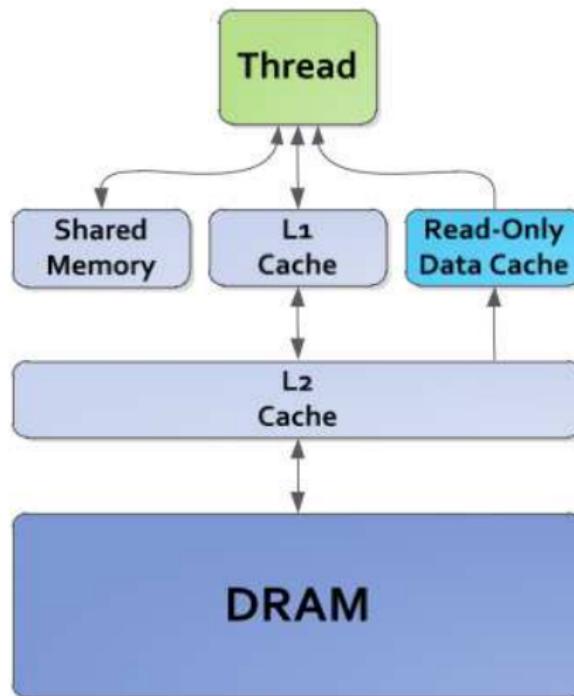
Kepler and Pascal

| GPU | Kepler GK110 | Maxwell GM200 | Pascal GP100 |
|------------------------------------|-------------------|---------------|--------------|
| Compute Capability | 3.5 | 5.2 | 6.0 |
| Threads / Warp | 32 | 32 | 32 |
| Max Warps / Multiprocessor | 64 | 64 | 64 |
| Max Threads / Multiprocessor | 2048 | 2048 | 2048 |
| Max Thread Blocks / Multiprocessor | 16 | 32 | 32 |
| Max 32-bit Registers / SM | 65536 | 65536 | 65536 |
| Max Registers / Block | 65536 | 32768 | 65536 |
| Max Registers / Thread | 255 | 255 | 255 |
| Max Thread Block Size | 1024 | 1024 | 1024 |
| Shared Memory Size / SM | 16 KB/32 KB/48 KB | 96 KB | 64 KB |

from the NVIDIA Tesla P100 Whitepaper

memory hierarchies

Kepler Memory Hierarchy



summary and references

To fully utilize the GPU, one must use thousands of threads, whereas running thousands of threads on multicore CPU will swamp it.

Data independence is the key difference between GPU and CPU.

We covered most of chapter 2 of the textbook by Kirk & Hwu.

NVIDIA. Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Fermi.

NVIDIA. Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110.

NVIDIA. Whitepaper NVIDIA Tesla P100.

Available at <http://www.nvidia.com>

Chapter 12 of the book of Wilkinson & Allen is on Image Processing.

Introduction to CUDA

1 Our first GPU Program

- running Newton's method in complex arithmetic
- examining the CUDA Compute Capability

2 CUDA Program Structure

- steps to write code for the GPU
- code to compute complex roots
- the kernel function and main program
- a scalable programming model

MCS 572 Lecture 30
Introduction to Supercomputing
Jan Verschelde, 31 October 2016

Introduction to CUDA

1 Our first GPU Program

- running Newton's method in complex arithmetic
- examining the CUDA Compute Capability

2 CUDA Program Structure

- steps to write code for the GPU
- code to compute complex roots
- the kernel function and main program
- a scalable programming model

computing complex square roots

To compute \sqrt{c} for $c \in \mathbb{C}$,
we apply Newton's method on $x^2 - c = 0$:

$$x_0 := c, \quad x_{k+1} := x_k - \frac{x_k^2 - c}{2x_k}, \quad k = 0, 1, \dots$$

Five iterations suffice to obtain an accurate value for \sqrt{c} .

Suitable on GPU?

- Finding roots is relevant for scientific computing.
- Data parallelism: compute for many different c 's.

Application: complex root finder for polynomials in one variable.

Introduction to CUDA

1 Our first GPU Program

- running Newton's method in complex arithmetic
- examining the CUDA Compute Capability

2 CUDA Program Structure

- steps to write code for the GPU
- code to compute complex roots
- the kernel function and main program
- a scalable programming model

CUDA Compute Capability

The compute capability of an NVIDIA GPU

- is represented by a version number in the format x.y,
- identifies the features supported by the hardware.

What does it mean for the programmer? Some examples:

1.3 : double-precision floating-point operations

2.0 : synchronizing threads

3.5 : dynamic parallelism

5.3 : half-precision floating-point operations

6.0 : atomic addition operation on 64-bit floats

The compute capability is not the same as the CUDA version.

checking the card with deviceQuery on kepler

```
$ /usr/local/cuda/samples/1_Utilities/deviceQuery/deviceQuery
/usr/local/cuda/samples/1_Utilities/deviceQuery/deviceQuery Starting...

CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 3 CUDA Capable device(s)

Device 0: "Tesla K20c"
  CUDA Driver Version / Runtime Version      6.0 / 5.5
  CUDA Capability Major/Minor version number: 3.5
  Total amount of global memory:             4800 MBytes (5032706048 bytes)
  (13) Multiprocessors, (192) CUDA Cores/MP: 2496 CUDA Cores
  GPU Clock rate:                          706 MHz (0.71 GHz)
  Memory Clock rate:                      2600 Mhz
  Memory Bus Width:                       320-bit
  L2 Cache Size:                           1310720 bytes
  Maximum Texture Dimension Size (x,y,z)    1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers 1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers 2D=(16384, 16384), 2048 layers
  Total amount of constant memory:           65536 bytes
  Total amount of shared memory per block:   49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                32
  Maximum number of threads per multiprocessor: 2048
  Maximum number of threads per block:        1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                     2147483647 bytes
  Texture alignment:                        512 bytes
  Concurrent copy and kernel execution:     Yes with 2 copy engine(s)
  Run time limit on kernels:                No
  Integrated GPU sharing Host Memory:       No
  Support host page-locked memory mapping: Yes
  Alignment requirement for Surfaces:       Yes
  Device has ECC support:                  Enabled
  Device supports Unified Addressing (UVA): Yes
  Device PCI Bus ID / PCI location ID:    4 / 0
  Compute Mode:
    < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```



checking the card with deviceQuery on pascal

```
$ /usr/local/cuda/samples/1_Utilities/deviceQuery/deviceQuery
/usr/local/cuda/samples/1_Utilities/deviceQuery/deviceQuery Starting...

CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 2 CUDA Capable device(s)

Device 0: "Tesla P100-PCIE-16GB"
  CUDA Driver Version / Runtime Version      8.0 / 8.0
  CUDA Capability Major/Minor version number: 6.0
  Total amount of global memory:             16276 MBytes (17066885120 bytes)
    (56) Multiprocessors, ( 64) CUDA Cores/MP:
  GPU Max Clock rate:                      405 MHz (0.41 GHz)
  Memory Clock rate:                       715 Mhz
  Memory Bus Width:                        4096-bit
  L2 Cache Size:                           4194304 bytes
  Maximum Texture Dimension Size (x,y,z)     1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers   1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers   2D=(32768, 32768), 2048 layers
  Total amount of constant memory:            65536 bytes
  Total amount of shared memory per block:     49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                32
  Maximum number of threads per multiprocessor: 2048
  Maximum number of threads per block:        1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size (x,y,z):  (2147483647, 65535, 65535)
  Maximum memory pitch:                      2147483647 bytes
  Texture alignment:                         512 bytes
  Concurrent copy and kernel execution:       Yes with 2 copy engine(s)
  Run time limit on kernels:                 No
  Integrated GPU sharing Host Memory:        No
  Support host page-locked memory mapping:   Yes
  Alignment requirement for Surfaces:        Yes
  Device has ECC support:                   Enabled
  Device supports Unified Addressing (UVA): Yes
  Device PCI Domain ID / Bus ID / location ID: 0 / 2 / 0
  Compute Mode:
    < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```



running bandwidthTest on kepler

```
$ /usr/local/cuda/samples/1_Utilities/bandwidthTest/bandwidthTest

[CUDA Bandwidth Test] - Starting...
Running on...

Device 0: Tesla K20c
Quick Mode

Host to Device Bandwidth, 1 Device(s)
PINNED Memory Transfers
Transfer Size (Bytes) Bandwidth(MB/s)
33554432 5819.5

Device to Host Bandwidth, 1 Device(s)
PINNED Memory Transfers
Transfer Size (Bytes) Bandwidth(MB/s)
33554432 6415.8

Device to Device Bandwidth, 1 Device(s)
PINNED Memory Transfers
Transfer Size (Bytes) Bandwidth(MB/s)
33554432 143248.0

Result = PASS
```

running bandwidthTest on pascal

```
$ /usr/local/cuda/samples/1_Utilities/bandwidthTest/bandwidthTest
[CUDA Bandwidth Test] - Starting...
Running on...

Device 0: Tesla P100-PCIE-16GB
Quick Mode

Host to Device Bandwidth, 1 Device(s)
PINNED Memory Transfers
Transfer Size (Bytes) Bandwidth(MB/s)
33554432 11530.1

Device to Host Bandwidth, 1 Device(s)
PINNED Memory Transfers
Transfer Size (Bytes) Bandwidth(MB/s)
33554432 12848.3

Device to Device Bandwidth, 1 Device(s)
PINNED Memory Transfers
Transfer Size (Bytes) Bandwidth(MB/s)
33554432 444598.8

Result = PASS

$
```

Introduction to CUDA

1 Our first GPU Program

- running Newton's method in complex arithmetic
- examining the CUDA Compute Capability

2 CUDA Program Structure

- **steps to write code for the GPU**
- code to compute complex roots
- the kernel function and main program
- a scalable programming model

steps to write code for the GPU

Five steps to get GPU code running:

- ① C and C++ functions are labeled with CUDA keywords
 `__device__`, `__global__`, or `__host__`.
- ② Determine the data for each thread to work on.
- ③ Transferring data from/to host (CPU) to/from the device (GPU).
- ④ Statements to launch data-parallel functions, called *kernels*.
- ⑤ Compilation with `nvcc`.

step 1: CUDA extensions to functions

Three keywords before a function declaration:

`__host__` : The function will run on the host (CPU).

`__device__` : The function will run on the device (GPU).

`__global__` : The function is called from the host but runs on the device. This function is called a *kernel*.

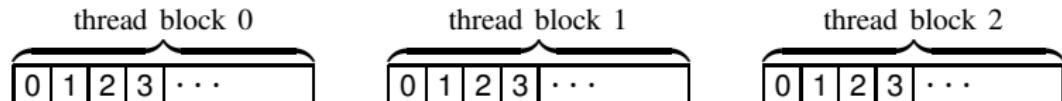
CUDA extensions to C function declarations:

| | executed on | callable from |
|------------------------------------|-------------|---------------|
| <code>__device__ double D()</code> | device | device |
| <code>__global__ void K()</code> | device | host |
| <code>__host__ int H()</code> | host | host |

step 2: data for each thread

The grid consists of N blocks, with $\text{blockIdx.x} \in \{0, N - 1\}$.

Within each block, $\text{threadIdx.x} \in \{0, \text{blockDim.x} - 1\}$.



```
int threadIdx = blockIdx.x *  
    blockDim.x + threadIdx.x  
...  
float x = input[threadID]  
float y = f(x)  
output[threadID] = y  
...
```

step 3: allocating and transferring data

```
cudaDoubleComplex *xhost = new cudaDoubleComplex[n];  
  
// we copy n complex numbers to the device  
size_t s = n*sizeof(cudaDoubleComplex);  
cudaDoubleComplex *xdevice;  
cudaMalloc((void**)&xdevice,s);  
  
cudaMemcpy(xdevice,xhost,s,cudaMemcpyHostToDevice);  
  
// allocate memory for the result  
cudaDoubleComplex *ydevice;  
cudaMalloc((void**)&ydevice,s);  
  
// copy results from device to host  
cudaDoubleComplex *yhost = new cudaDoubleComplex[n];  
  
cudaMemcpy(yhost,ydevice,s,cudaMemcpyDeviceToHost);
```

step 4: launching the kernel

The kernel is declared as

```
__global__ void squareRoot
( int n, cudaDoubleComplex *x, cudaDoubleComplex *y )
// Applies Newton's method to compute the square root
// of the n numbers in x and places the results in y.
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    ...
}
```

For frequency f , dimension n , and block size w , we do:

```
// invoke the kernel with n/w blocks per grid
// and w threads per block
for(int i=0; i<f; i++)
    squareRoot<<<n/w, w>>>(n, xdevice, ydevice);
```

step 5: compiling with nvcc

Then if the makefile contains

```
runCudaComplexSqrt:  
    nvcc -o /tmp/run_cmplxsqrt -arch=sm_13 \  
          runCudaComplexSqrt.cu
```

typing make runCudaComplexSqrt does

```
nvcc -o /tmp/run_cmplxsqrt -arch=sm_13 runCudaComplexSqrt.cu
```

The -arch=sm_13 is needed for double arithmetic, for the K20C.

The option -arch=sm_13 is no longer recognized on the new P100.

Introduction to CUDA

1 Our first GPU Program

- running Newton's method in complex arithmetic
- examining the CUDA Compute Capability

2 CUDA Program Structure

- steps to write code for the GPU
- **code to compute complex roots**
- the kernel function and main program
- a scalable programming model

defining complex numbers

```
#ifndef __CUDADOUBLECOMPLEX_CU__
#define __CUDADOUBLECOMPLEX_CU__

#include <cmath>
#include <cstdlib>
#include <iomanip>
#include <vector_types.h>
#include <math_functions.h>

typedef double2 cudaDoubleComplex;
```

We use the `double2` of `vector_types.h` to define complex numbers because `double2` is a native CUDA type allowing for coalesced memory access.

random complex numbers

```
__host__ cudaDoubleComplex randomDoubleComplex()
// Returns a complex number on the unit circle
// with angle uniformly generated in [0,2*pi].
{
    cudaDoubleComplex result;
    int r = rand();
    double u = double(r)/RAND_MAX;
    double angle = 2.0*M_PI*u;
    result.x = cos(angle);
    result.y = sin(angle);
    return result;
}
```

calling sqrt of math_functions.h

```
__device__ double radius ( const cudaDoubleComplex c )
// Returns the radius of the complex number.
{
    double result;
    result = c.x*c.x + c.y*c.y;
    return sqrt(result);
}
```

overloading for output

```
__host__ std::ostream& operator<<
( std::ostream& os, const cudaDoubleComplex& c)
// Writes real and imaginary parts of c,
// in scientific notation with precision 16.
{
    os << std::scientific << std::setprecision(16)
        << c.x << " " << c.y;
    return os;
}
```

defining complex addition

```
__device__ cudaDoubleComplex operator+
( const cudaDoubleComplex a, const cudaDoubleComplex b )
// Returns the sum of a and b.
{
    cudaDoubleComplex result;
    result.x = a.x + b.x;
    result.y = a.y + b.y;
    return result;
}
```

The rest of the arithmetical operations are defined in a similar manner.

All definitions related to complex numbers are stored in the file `cudaDoubleComplex.cu`.

Introduction to CUDA

1 Our first GPU Program

- running Newton's method in complex arithmetic
- examining the CUDA Compute Capability

2 CUDA Program Structure

- steps to write code for the GPU
- code to compute complex roots
- the kernel function and main program**
- a scalable programming model

the kernel function

```
#include "cudaDoubleComplex.cu"

__global__ void squareRoot
( int n, cudaDoubleComplex *x, cudaDoubleComplex *y )
// Applies Newton's method to compute the square root
// of the n numbers in x and places the results in y.
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    cudaDoubleComplex inc;
    cudaDoubleComplex c = x[i];
    cudaDoubleComplex r = c;
    for(int j=0; j<5; j++)
    {
        inc = r + r;
        inc = (r*r - c)/inc;
        r = r - inc;
    }
    y[i] = r;
}
```

the main function — command line arguments

```
int main ( int argc, char*argv[] )  
{  
    if(argc < 5)  
    {  
        cout << "call with 4 arguments : " << endl;  
        cout << "dimension, block size, frequency, and check (0 or 1)"  
            << endl;  
    }  
    else  
    {  
        int n = atoi(argv[1]); // dimension  
        int w = atoi(argv[2]); // block size  
        int f = atoi(argv[3]); // frequency  
        int t = atoi(argv[4]); // test or not  
        // we generate n random complex numbers on the host  
        cudaDoubleComplex *xhost = new cudaDoubleComplex[n];  
        for(int i=0; i<n; i++) xhost[i] = randomDoubleComplex();  
    }  
}
```

The main program generates n random complex numbers with radius 1.

transferring data and launching the kernel

```
// copy the n random complex numbers to the device
size_t s = n*sizeof(cudaDoubleComplex);
cudaDoubleComplex *xdevice;
cudaMalloc((void**)&xdevice,s);
cudaMemcpy(xdevice,xhost,s,cudaMemcpyHostToDevice);
// allocate memory for the result
cudaDoubleComplex *ydevice;
cudaMalloc((void**)&ydevice,s);
// invoke the kernel with n/w blocks per grid
// and w threads per block
for(int i=0; i<f; i++)
    squareRoot<<<n/w,w>>>(n,xdevice,ydevice);
// copy results from device to host
cudaDoubleComplex *yhost = new cudaDoubleComplex[n];
cudaMemcpy(yhost,ydevice,s,cudaMemcpyDeviceToHost);
```

testing one random number

```
if(t == 1) // test the result
{
    int k = rand() % n;
    cout << "testing number " << k << endl;
    cout << "           x = " << xhost[k] << endl;
    cout << " sqrt(x) = " << yhost[k] << endl;
    cudaDoubleComplex z = Square(yhost[k]);
    cout << "sqrt(x)^2 = " << z << endl;
}
return 0;
}
```

Introduction to CUDA

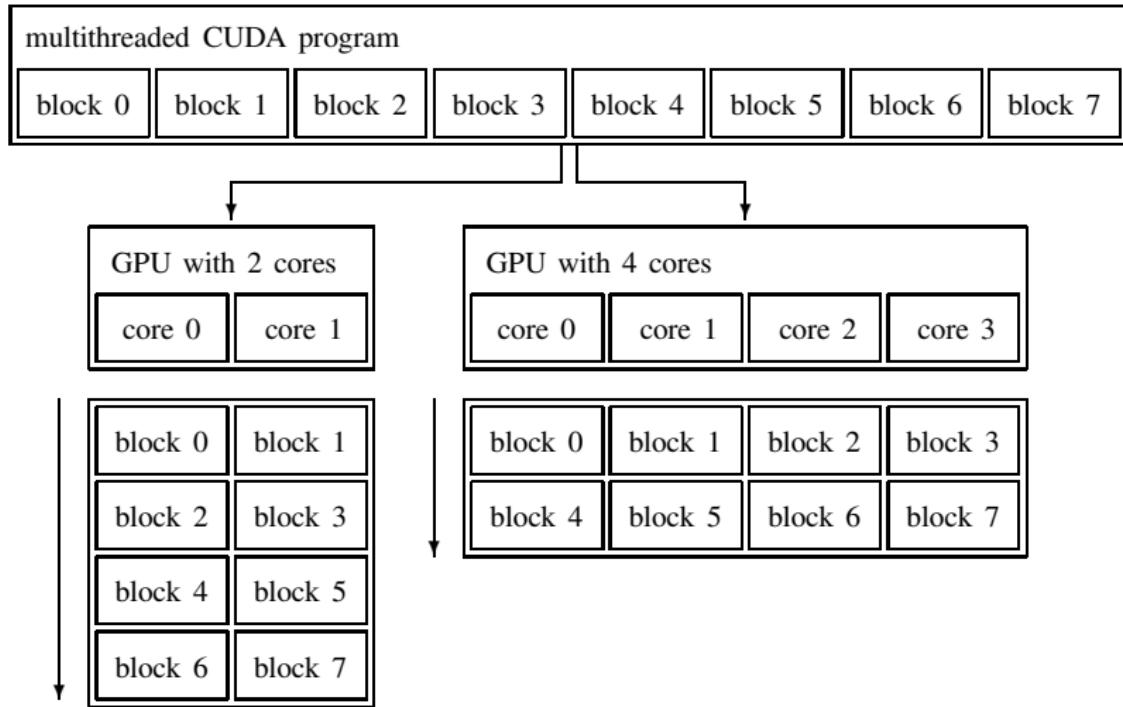
1 Our first GPU Program

- running Newton's method in complex arithmetic
- examining the CUDA Compute Capability

2 CUDA Program Structure

- steps to write code for the GPU
- code to compute complex roots
- the kernel function and main program
- **a scalable programming model**

a scalable programming model



running the code on kepler

A test on the correctness:

```
$ ./tmp/run_cmpsqrt 1 1 1 1  
testing number 0  
    x = 5.3682227446949737e-01 -8.4369535119816541e-01  
sqrt(x) = 8.7659063264145631e-01 -4.8123680528950746e-01  
sqrt(x)^2 = 5.3682227446949726e-01 -8.4369535119816530e-01
```

On 64,000 numbers, 32 threads in a block, doing it 10,000 times:

```
$ time ./tmp/run_cmpsqrt 64000 32 10000 1  
testing number 50325  
    x = 7.9510606509728776e-01 -6.0647039931517477e-01  
sqrt(x) = 9.4739275517002119e-01 -3.2007337822967424e-01  
sqrt(x)^2 = 7.9510606509728765e-01 -6.0647039931517477e-01  
  
real    0m1.618s  
user    0m0.526s  
sys     0m0.841s  
$
```

changing #threads in a block

```
$ time /tmp/run_cmpsqrt 128000 32 100000 0
```

```
real      0m17.345s  
user      0m9.829s  
sys       0m7.303s
```

```
$ time /tmp/run_cmpsqrt 128000 64 100000 0
```

```
real      0m10.502s  
user      0m5.711s  
sys       0m4.497s
```

```
$ time /tmp/run_cmpsqrt 128000 128 100000 0
```

```
real      0m9.295s  
user      0m5.231s  
sys       0m3.865s
```

running the code on pascal

```
$ time /tmp/run_cmpsqrt 128000 32 100000 0
```

```
real      0m2.516s  
user      0m1.250s  
sys       0m1.236s
```

```
$ time /tmp/run_cmpsqrt 128000 64 100000 0
```

```
real      0m2.521s  
user      0m1.245s  
sys       0m1.234s
```

```
$ time /tmp/run_cmpsqrt 128000 128 100000 0
```

```
real      0m2.496s  
user      0m1.288s  
sys       0m1.195s
```

summary and references

In five steps we wrote our first complete CUDA program.

We started chapter 3 of the textbook by Kirk & Hwu,
covering more of the CUDA Programming Guide.

Available in `/usr/local/cuda/doc` are

- CUDA C Best Practices Guide
- CUDA Programming Guide

Also available online at nvidia.com.

Many examples of CUDA applications are available in
`/usr/local/cuda/samples`.

exercises

- 1 Instead of 5 Newton iterations in `runCudaComplexSqrt.cu` use k iterations where k is entered by the user at the command line. What is the influence of k on the timings?
- 2 Modify the kernel for the complex square root so it takes on input an array of complex coefficients of a polynomial of degree d . Then the root finder applies Newton's method, starting at random points. Test the correctness and experiment to find the rate of success, i.e.: for polynomials of degree d how many random trials are needed to obtain $d/2$ roots of the polynomial?

Data Parallelism and Matrix Multiplication

1 Data Parallelism

- matrix-matrix multiplication
- CUDA program structure

2 Code for Matrix-Matrix Multiplication

- linear address system for 2-dimensional array
- defining the kernel
- the main program

3 Two Dimensional Arrays of Threads

- using `threadIdx.x` and `threadIdx.y`

4 Examining Performance

- counting flops

MCS 572 Lecture 31
Introduction to Supercomputing
Jan Verschelde, 2 November 2016

Data Parallelism and Matrix Multiplication

1 Data Parallelism

- matrix-matrix multiplication
- CUDA program structure

2 Code for Matrix-Matrix Multiplication

- linear address system for 2-dimensional array
- defining the kernel
- the main program

3 Two Dimensional Arrays of Threads

- using `threadIdx.x` and `threadIdx.y`

4 Examining Performance

- counting flops

data parallelism

Many applications process large amounts of data.

Data parallelism refers to the property where many arithmetic operations can be safely performed on the data simultaneously.

Consider the multiplication of matrices A and B : $C = A \cdot B$, with

$$A = [a_{i,j}] \in \mathbb{R}^{n \times m}, \quad B = [b_{i,j}] \in \mathbb{R}^{m \times p}, \quad C = [c_{i,j}] \in \mathbb{R}^{n \times p}.$$

$c_{i,j}$ is the inner product of the i th row of A with the j th column of B :

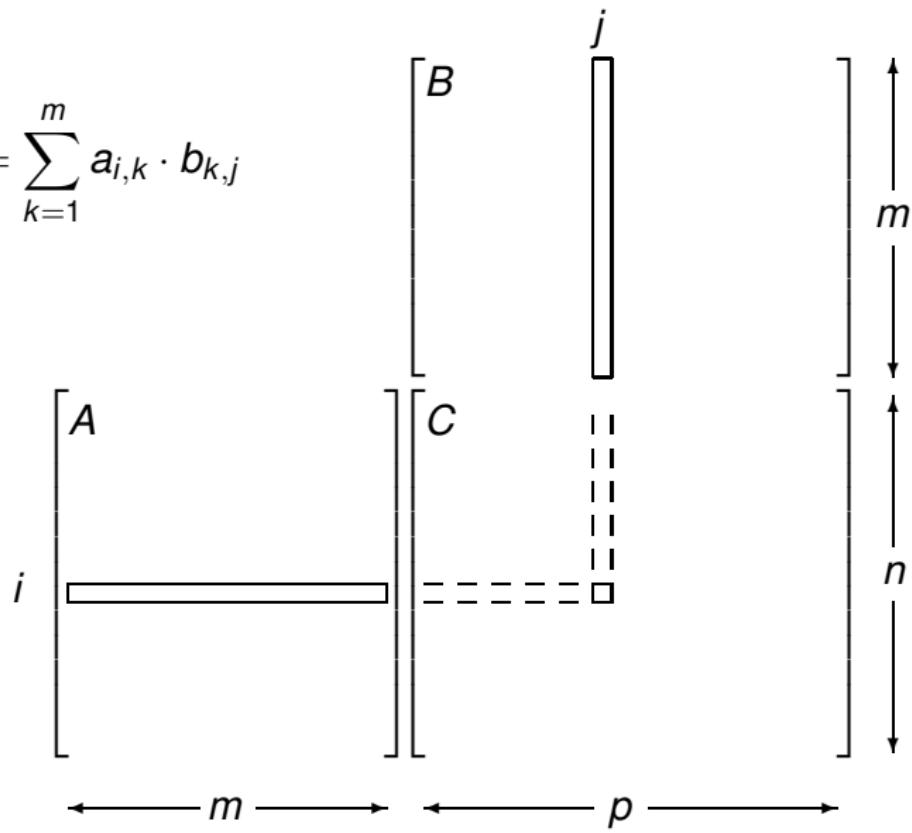
$$c_{i,j} = \sum_{k=1}^m a_{i,k} \cdot b_{k,j}.$$

All $c_{i,j}$'s can be computed independently from each other.

For $n = m = p = 1,000$ we have 1,000,000 inner products.

data parallelism in matrix multiplication

$$c_{i,j} = \sum_{k=1}^m a_{i,k} \cdot b_{k,j}$$



matrix-matrix multiplication on a GPU

Code for a device (the GPU) is defined in functions using the keyword `__global__` before the function definition.

Data parallel functions are called *kernels*.

Kernel functions generate a large number of threads.

In matrix-matrix multiplication, the computation can be implemented as a kernel where each thread computes one element in the result matrix.

To multiply two 1,000-by-1,000 matrices, the kernel using one thread to compute one element generates 1,000,000 threads when invoked.

CUDA threads are much lighter weight than CPU threads: they take very few cycles to generate and schedule thanks to efficient hardware support whereas CPU threads may require thousands of cycles.

Data Parallelism and Matrix Multiplication

1 Data Parallelism

- matrix-matrix multiplication
- CUDA program structure

2 Code for Matrix-Matrix Multiplication

- linear address system for 2-dimensional array
- defining the kernel
- the main program

3 Two Dimensional Arrays of Threads

- using `threadIdx.x` and `threadIdx.y`

4 Examining Performance

- counting flops

CUDA program structure

A CUDA program consists of several phases, executed on

- the host: if no data parallelism,
- the device: for data parallel algorithms.

The NVIDIA C compiler `nvcc` separates phases at compilation:

- Code for the host is compiled on host's standard C compilers and runs as ordinary CPU process.
- The device code is written in C with keywords for data parallel functions and further compiled by `nvcc`.

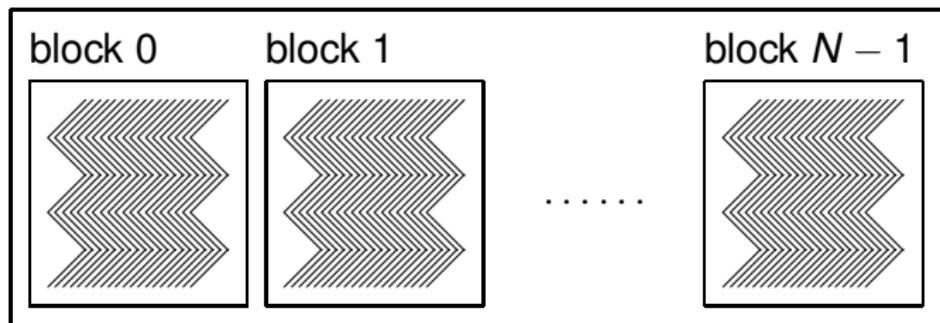
execution of a CUDA program

CPU code

```
kernel<<<numb_blocks, numb_threads_per_block>>>(args)
```

CPU code

grid



stages in a CUDA program

For the matrix multiplication $C = A \cdot B$:

- ① Allocate device memory for A , B , and C .
- ② Copy A and B from the host to the device.
- ③ Invoke the kernel to have device do $C = A \cdot B$.
- ④ Copy C from the device to the host.
- ⑤ Free memory space on the device.

Data Parallelism and Matrix Multiplication

1 Data Parallelism

- matrix-matrix multiplication
- CUDA program structure

2 Code for Matrix-Matrix Multiplication

- linear address system for 2-dimensional array
- defining the kernel
- the main program

3 Two Dimensional Arrays of Threads

- using `threadIdx.x` and `threadIdx.y`

4 Examining Performance

- counting flops

linear address system

Consider a 3-by-5 matrix stored row-wise (as in C):

| | | | | |
|-----------|-----------|-----------|-----------|-----------|
| $a_{0,0}$ | $a_{0,1}$ | $a_{0,2}$ | $a_{0,3}$ | $a_{0,4}$ |
| $a_{1,0}$ | $a_{1,1}$ | $a_{1,2}$ | $a_{1,3}$ | $a_{1,4}$ |
| $a_{2,0}$ | $a_{2,1}$ | $a_{2,2}$ | $a_{2,3}$ | $a_{2,4}$ |



| | | | | | | | | | | | | | | |
|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| $a_{0,0}$ | $a_{0,1}$ | $a_{0,2}$ | $a_{0,3}$ | $a_{0,4}$ | $a_{1,0}$ | $a_{1,1}$ | $a_{1,2}$ | $a_{1,3}$ | $a_{1,4}$ | $a_{2,0}$ | $a_{2,1}$ | $a_{2,2}$ | $a_{2,3}$ | $a_{2,4}$ |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

We will store a matrix as a one dimensional array.

generating a random matrix

```
#include <stdlib.h>

__host__ void randomMatrix ( int n, int m, float *x, int mode )
/*
 * Fills up the n-by-m matrix x with random
 * values of zeroes and ones if mode == 1,
 * or random floats if mode == 0. */
{
    int i,j,r;
    float *p = x;

    for(i=0; i<n; i++)
        for(j=0; j<m; j++)
    {
        if(mode == 1)
            r = rand() % 2;
        else
            r = ((float) rand()) / RAND_MAX;
        *(p++) = (float) r;
    }
}
```

writing a matrix

```
#include <stdio.h>

__host__ void writeMatrix ( int n, int m, float *x )
/*
 * Writes the n-by-m matrix x to screen. */
{
    int i,j;
    float *p = x;

    for(i=0; i<n; i++,printf("\n"))
        for(j=0; j<m; j++)
            printf(" %d", (int)* (p++));
}
```

Data Parallelism and Matrix Multiplication

1 Data Parallelism

- matrix-matrix multiplication
- CUDA program structure

2 Code for Matrix-Matrix Multiplication

- linear address system for 2-dimensional array
- defining the kernel**
- the main program

3 Two Dimensional Arrays of Threads

- using `threadIdx.x` and `threadIdx.y`

4 Examining Performance

- counting flops

assigning inner products to threads

Consider a 3-by-4 matrix A and a 4-by-5 matrix B :

| | | | |
|-----------|-----------|-----------|-----------|
| $a_{0,0}$ | $a_{0,1}$ | $a_{0,2}$ | $a_{0,3}$ |
| $a_{1,0}$ | $a_{1,1}$ | $a_{1,2}$ | $a_{1,3}$ |
| $a_{2,0}$ | $a_{2,1}$ | $a_{2,2}$ | $a_{2,3}$ |

| | | | | |
|-----------|-----------|-----------|-----------|-----------|
| $b_{0,0}$ | $b_{0,1}$ | $b_{0,2}$ | $b_{0,3}$ | $b_{0,4}$ |
| $b_{1,0}$ | $b_{1,1}$ | $b_{1,2}$ | $b_{1,3}$ | $b_{1,4}$ |
| $b_{2,0}$ | $b_{2,1}$ | $b_{2,2}$ | $b_{2,3}$ | $b_{2,4}$ |
| $b_{3,0}$ | $b_{3,1}$ | $b_{3,2}$ | $b_{3,3}$ | $b_{3,4}$ |

| | | | | | | | | | | | | | | |
|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| $c_{0,0}$ | $c_{0,1}$ | $c_{0,2}$ | $c_{0,3}$ | $c_{0,4}$ | $c_{1,0}$ | $c_{1,1}$ | $c_{1,2}$ | $c_{1,3}$ | $c_{1,4}$ | $c_{2,0}$ | $c_{2,1}$ | $c_{2,2}$ | $c_{2,3}$ | $c_{2,4}$ |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

The $i = blockIdx.x * blockDim.x + threadIdx.x$ determines what entry in $C = A \cdot B$ will be computed:

- the row index in C is i divided by 5 and
- the column index in C is the remainder of i divided by 5.

the kernel function

```
__global__ void matrixMultiply
( int n, int m, int p, float *A, float *B, float *C )
/*
 * Multiplies the n-by-m matrix A
 * with the m-by-p matrix B into the matrix C.
 * The i-th thread computes the i-th element of C. */
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    C[i] = 0.0;
    int rowC = i/p;
    int colC = i%p;
    float *pA = &A[rowC*m];
    float *pB = &B[colC];
    for(int k=0; k<m; k++)
    {
        pB = &B[colC+k*p];
        C[i] += (* (pA++)) * (*pB);
    }
}
```

Data Parallelism and Matrix Multiplication

1 Data Parallelism

- matrix-matrix multiplication
- CUDA program structure

2 Code for Matrix-Matrix Multiplication

- linear address system for 2-dimensional array
- defining the kernel
- the main program**

3 Two Dimensional Arrays of Threads

- using `threadIdx.x` and `threadIdx.y`

4 Examining Performance

- counting flops

running the program

```
$ ./tmp/matmatmul 3 4 5 1
a random 3-by-4 0/1 matrix A :
1 0 1 1
1 1 1 1
1 0 1 0
a random 4-by-5 0/1 matrix B :
0 1 0 0 1
0 1 1 0 0
1 1 0 0 0
1 1 0 1 0
the resulting 3-by-5 matrix C :
2 3 0 1 1
2 4 1 1 1
1 2 0 0 1
$
```

the main program — command line arguments

```
int main ( int argc, char*argv[] )  
{  
    if(argc < 4)  
    {  
        printf("call with 3 arguments :\\n");  
        printf("dimensions n, m, and p\\n");  
    }  
    else  
    {  
        int n = atoi(argv[1]);      /* number of rows of A */  
        int m = atoi(argv[2]);      /* number of columns of A */  
                                /* and number of rows of B */  
        int p = atoi(argv[3]);      /* number of columns of B */  
        int mode = atoi(argv[4]); /* 0 no output, 1 show output */  
        if(mode == 0)  
            srand(20140331)  
        else  
            srand(time(0));
```

allocating memories

```
float *Ahost = (float*)calloc(n*m,sizeof(float));
float *Bhost = (float*)calloc(m*p,sizeof(float));
float *Chost = (float*)calloc(n*p,sizeof(float));
randomMatrix(n,m,Ahost,mode);
randomMatrix(m,p,Bhost,mode);
if(mode == 1)
{
    printf("a random %d-by-%d 0/1 matrix A :\n",n,m);
    writeMatrix(n,m,Ahost);
    printf("a random %d-by-%d 0/1 matrix B :\n",m,p);
    writeMatrix(m,p,Bhost);
}
/* allocate memory on the device for A, B, and C */
float *Adevice;
size_t sA = n*m*sizeof(float);
cudaMalloc((void**)&Adevice,sA);
float *Bdevice;
size_t sB = m*p*sizeof(float);
cudaMalloc((void**)&Bdevice,sB);
float *Cdevice;
size_t sC = n*p*sizeof(float);
cudaMalloc((void**)&Cdevice,sC);
```

copying and kernel invocation

```
/* copy matrices A and B from host to the device */
cudaMemcpy(Adevice,Ahost,sA,cudaMemcpyHostToDevice);
cudaMemcpy(Bdevice,Bhost,sB,cudaMemcpyHostToDevice);

/* kernel invocation launching n*p threads */
matrixMultiply<<<n*p,1>>>(n,m,p,
                                Adevice,Bdevice,Cdevice);

/* copy matrix C from device to the host */
cudaMemcpy(Chost,Cdevice,sC,cudaMemcpyDeviceToHost);
/* freeing memory on the device */
cudaFree(Adevice); cudaFree(Bdevice); cudaFree(Cdevice);
if(mode == 1)
{
    printf("the resulting %d-by-%d matrix C :\n",n,p);
    writeMatrix(n,p,Chost);
}
return 0;
}
```

Data Parallelism and Matrix Multiplication

1 Data Parallelism

- matrix-matrix multiplication
- CUDA program structure

2 Code for Matrix-Matrix Multiplication

- linear address system for 2-dimensional array
- defining the kernel
- the main program

3 Two Dimensional Arrays of Threads

- using `threadIdx.x` and `threadIdx.y`

4 Examining Performance

- counting flops

using `threadIdx.x` and `threadIdx.y`

Instead of a one dimensional organization of the threads in a block we can make the (i, j) -th thread compute $c_{i,j}$.

The main program is then changed into

```
/* kernel invocation launching n*p threads */
dim3 dimGrid(1,1);
dim3 dimBlock(n,p);
matrixMultiply<<<dimGrid, dimBlock>>>
    (n, m, p, Adevice, Bdevice, Cdevice);
```

The above construction creates a grid of one block.

The block has $n \times p$ threads:

- `threadIdx.x` will range between 0 and $n - 1$, and
- `threadIdx.y` will range between 0 and $p - 1$.

the new kernel

```
__global__ void matrixMultiply
( int n, int m, int p, float *A, float *B, float *C )
/*
 * Multiplies the n-by-m matrix A
 * with the m-by-p matrix B into the matrix C.
 * The (i,j)-th thread computes the (i,j)-th element of C.
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    int ell = i*p + j;
    C[ell] = 0.0;
    float *pB;
    for(int k=0; k<m; k++)
    {
        pB = &B[j+k*p];
        C[ell] += A[i*m+k] * (*pB);
    }
}
```

Data Parallelism and Matrix Multiplication

1 Data Parallelism

- matrix-matrix multiplication
- CUDA program structure

2 Code for Matrix-Matrix Multiplication

- linear address system for 2-dimensional array
- defining the kernel
- the main program

3 Two Dimensional Arrays of Threads

- using `threadIdx.x` and `threadIdx.y`

4 Examining Performance

- counting flops

performance analysis

Performance is often expressed in terms of flops.

- 1 flops = one floating-point operation per second;
- use `perf`: Performance analysis tools for Linux
- run the executable, with `perf stat -e`
- with the events following the `-e` flag
we count the floating-point operations.

For the Intel Sandy Bridge in `kepler` the codes are

- 530110 : FP_COMP_OPS_EXE:X87
- 531010 : FP_COMP_OPS_EXE:SSE_FP_PACKED_DOUBLE
- 532010 : FP_COMP_OPS_EXE:SSE_FP_SCALAR_SINGLE
- 534010 : FP_COMP_OPS_EXE:SSE_PACKED_SINGLE
- 538010 : FP_COMP_OPS_EXE:SSE_SCALAR_DOUBLE

Executables are compiled with the option `-O2`.

performance of one CPU core

```
$ perf stat -e r530110 -e r531010 -e r532010 -e r534010 \
-e r538010 ./matmatmul0 745 745 745 0
```

```
Performance counter stats for './matmatmul0 745 745 745 0':
```

```
    1,668,710 r530110
```

```
        0 r531010
```

```
    2,478,340,803 r532010
```

```
        0 r534010
```

```
        0 r538010
```

```
1.033291591 seconds time elapsed
```

§

Did 2,480,009,513 operations in 1.033 seconds:

$$\Rightarrow (2,480,009,513 / 1.033) / (2^{30}) = 2.23 \text{GFlops.}$$

performance on the K20C

```
$ perf stat -e r530110 -e r531010 -e r532010 -e r534010 \
-e r538010 ./matmatmull 745 745 745 0

Performance counter stats for './matmatmull 745 745 745 0':


    160,925 r530110
                0 r531010
  2,306,222 r532010
                0 r534010
                0 r538010

  0.663709965 seconds time elapsed

$ time ./matmatmull 745 745 745 0

real      0m0.631s
user      0m0.023s
sys       0m0.462s
```

The dimension 745 is too small for the GPU to be able to improve much.

increasing the dimension

```
$ perf stat -e r530110 -e r531010 -e r532010 -e r534010 \
-e r538010 ./matmatmul0 4000 4000 4000 0

Performance counter stats for './matmatmul0 4000 4000 4000 0':
```

| | |
|-----------------|---------|
| 48,035,278 | r530110 |
| 0 | r531010 |
| 267,810,771,301 | r532010 |
| 0 | r534010 |
| 0 | r538010 |

171.334443720 seconds time elapsed

\$

See if we can speedup the computations with the GPU...

running on the K20C

```
$ perf stat -e r530110 -e r531010 -e r532010 -e r534010  
      -e r538010 ./matmatmul 4000 4000 4000 0  
  
Performance counter stats for './matmatmul 4000 4000 4000 0':  
  
    207,682 r530110  
          0 r531010  
 64,222,441 r532010  
          0 r534010  
          0 r538010  
  
 1.011284551 seconds time elapsed
```

\$

Speedup: $171.334/1.011 = 169.$

Counting flops, $f = 267,810,771,301$

- $t_{\text{cpu}} = 171.334: f/t_{\text{cpu}}/(2^{30}) = 1.5 \text{ GFlops.}$
- $t_{\text{gpu}} = 1.011: f/t_{\text{gpu}}/(2^{30}) = 246.7 \text{ GFlops.}$

running on pascal, on the P100

On a larger GPU, we need to scale the problem:

| $n = 2^k$ | n | time |
|-----------|-------|--------|
| 2^{12} | 4096 | 1.33s |
| 2^{13} | 8192 | 2.32s |
| 2^{14} | 16384 | 6.24s |
| 2^{15} | 32768 | 22.76s |

Matrix-Matrix multiplication is an $O(n^3)$ operation:
doubling the dimension, we expect the time increase eightfold.

A very rough estimate on the flops count (single precision floats):

- For $n = 4,000$, the flop count is $f = 267,810,771,301$.
- To scale to $n = 2^{15}$: $n \times 8$, so $F = 512 \times f$.
- In 22.76 seconds, so flops is $(F/22.76)/(2^{30}) = 5610.8$.

So we estimate the performance at 5.6 TeraFlops for $n = 32,000$.

summary and exercises

We covered more of chapter 3 in the book of Kirk & Hwu.

- ① Modify `matmatmul0.c` and `matmatmul1.cu` to work with doubles instead of floats. Examine the performance.
- ② Modify `matmatmul2.cu` to use double indexing of matrices, e.g.: $C[i][j] += A[i][k] * B[k][j]$.
- ③ Compare the performance of `matmatmul1.cu` and `matmatmul2.cu`, taking larger and larger values for n , m , and p . Which version scales best?

Device Memories and Matrix Multiplication

1 Device Memories

- global, constant, and shared memories
- CUDA variable type qualifiers

2 Matrix Multiplication

- an application of tiling
- running `matrixMul` in the GPU Computing SDK
- the kernel of `matrixMul`

MCS 572 Lecture 32
Introduction to Supercomputing
Jan Verschelde, 4 November 2016

Device Memories and Matrix Multiplication

1

Device Memories

- global, constant, and shared memories
- CUDA variable type qualifiers

2

Matrix Multiplication

- an application of tiling
- running `matrixMul` in the GPU Computing SDK
- the kernel of `matrixMul`

data storage on a graphics card

Before we launch a kernel, we have

- to allocate memory on the device,
- to transfer data from the host to the device.

By default, memory on the device is *global memory*.

In addition to global memory, we distinguish between

- registers for storing local variables,
- shared memory for all threads in a block,
- constant memory for all blocks on a grid.

compute to global memory access (CGMA) ratio

The importance of understanding different memories is in the calculation of the expected performance level of kernel code.

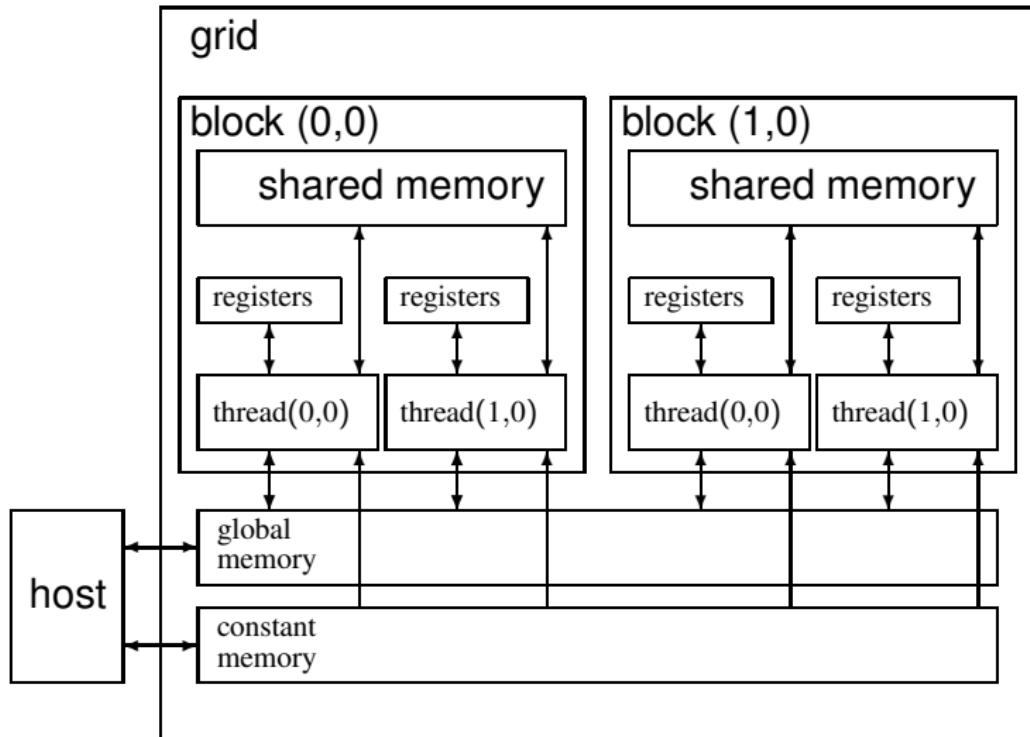
Definition

The *Compute to Global Memory Access (CGMA) ratio* is the number of floating-point calculations performed for each access to the global memory within a region of a CUDA program.

If the CGMA ratio is 1.0, then the memory clock rate determines the upper limit for the performance.

While memory bandwidth on a GPU is superior to that of a CPU, we will miss the theoretical peak performance by a factor of ten.

CUDA device memory types



registers

Registers are allocated to individual threads.
Each thread can access only its own registers.

A kernel function typically uses registers to hold frequently accessed variables that are private to each thread.

Number of 32-bit registers available per block:

- 8,192 on the GeForce 9400M,
- 32,768 on the Tesla C2050/C2070,
- 65,536 on the Tesla K20C and the P100.

A typical CUDA kernel may launch thousands of threads.

However, having too many local variables in a kernel function may prevent all blocks from running in parallel.

shared memory

Like registers, shared memory is an on-chip memory.

Variables residing in registers and shared memory can be accessed at very high speed in a highly parallel manner.

Unlike registers, which are private to each thread,
all threads in the same block have access to shared memory.

Amount of shared memory per block:

- 16,384 bytes on the GeForce 9400M,
- 49,152 bytes on the Tesla C2050/C2070,
- 49,152 bytes on the Tesla K20c and the P100.

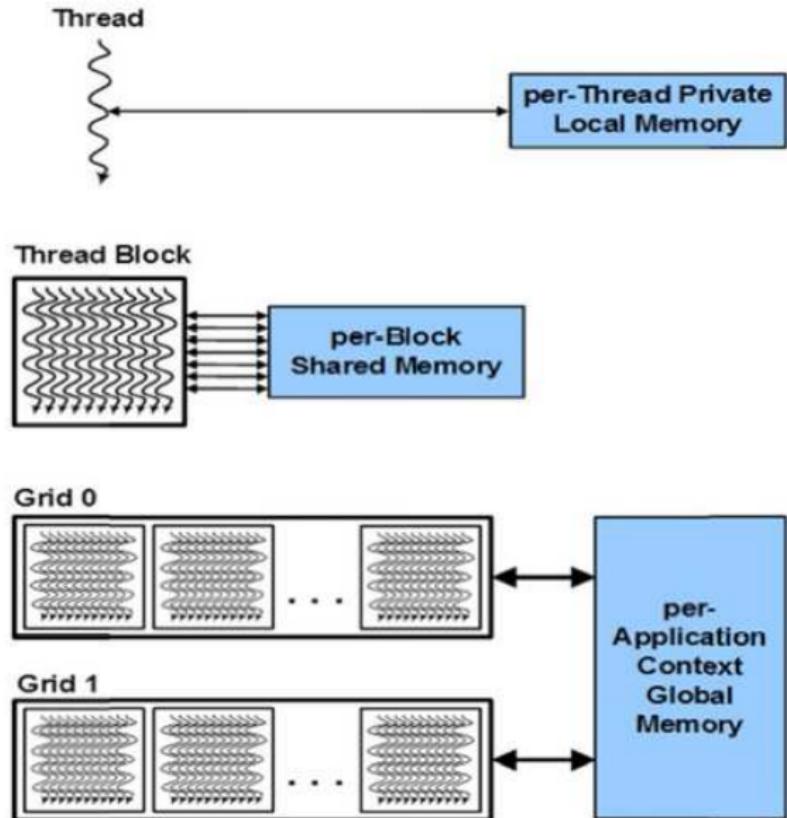
constant, global, and cache memory

The constant memory supports short-latency, high-bandwidth, read-only access by the device when all threads simultaneously access the same location.

Global memory is similar to RAM on the CPU.

| GPU | constant | global | L2 cache |
|---------------|----------|-----------|-------------|
| GeForce 9400M | 65,536 b | 254 Mb | |
| Tesla C2050 | 65,536 b | 2,687 Mb | 786,432 b |
| Tesla K20C | 65,536 b | 4,800 Mb | 1,310,720 b |
| Tesla P100 | 65,536 b | 16,276 Mb | 4,194,304 b |

a quick refresher



copied from the NVIDIA Whitepaper on Kepler GK110



Device Memories and Matrix Multiplication

1

Device Memories

- global, constant, and shared memories
- CUDA variable type qualifiers

2

Matrix Multiplication

- an application of tiling
- running `matrixMul` in the GPU Computing SDK
- the kernel of `matrixMul`

variables in memory, scope, and lifetime

Each variable is stored in a particular type of memory, has a scope and a lifetime.

Scope is the range of threads that can access the variable.

- If the scope of a variable is a single thread, then a private version of that variable exists for every single thread.
- Each thread can access only its private version of the variable.

Lifetime specifies the portion of the duration of the program execution when the variable is available for use.

- If a variable is declared in the kernel function body, then that variable is available for use only by the code of the kernel.
- If the kernel is invoked several times, then the contents of that variable will not be maintained across these invocations.

CUDA variable type qualifiers

We distinguish between five different variable declarations, based on their memory location, scope, and lifetime.

| variable declaration | memory | scope | lifetime |
|---|----------|--------|----------|
| atomic variables \neq arrays | register | thread | kernel |
| array variables | local | thread | kernel |
| <code>__device__ __shared__ .int v</code> | shared | block | kernel |
| <code>__device__ .int v</code> | global | grid | program |
| <code>__device__ __constant__ .int v</code> | constant | grid | program |

The `__device__` in front of `__shared__` is optional.

Device Memories and Matrix Multiplication

1

Device Memories

- global, constant, and shared memories
- CUDA variable type qualifiers

2

Matrix Multiplication

- an application of tiling
- running `matrixMul` in the GPU Computing SDK
- the kernel of `matrixMul`

the CGMA ratio

In our simple matrix-matrix multiplication $C = A \cdot B$, we have the statement

```
C[i] += (*pA++) * (*pB);
```

where

- C is a float array; and
- pA and pB are pointers to elements in a float array.

For the statement above, the CGMA ratio is 2/3:

- for one addition and one multiplication,
- we have three memory accesses.

an application of tiling

For $A \in \mathbb{R}^{n \times m}$ and $B \in \mathbb{R}^{m \times p}$, the product $C = A \cdot B \in \mathbb{R}^{n \times p}$.

Assume that n , m , and p are multiples of some w , e.g.: $w = 8$.

We compute C in tiles of size $w \times w$:

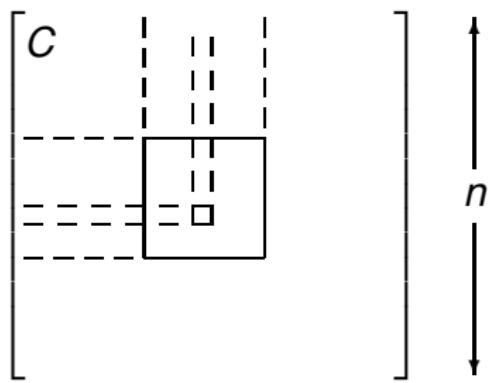
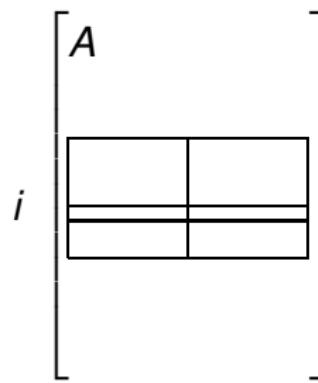
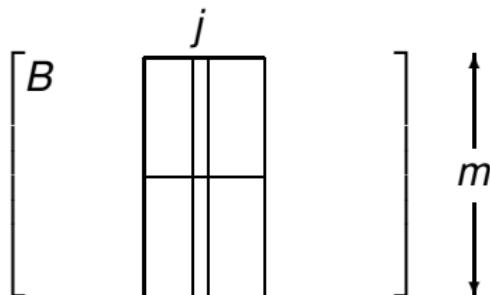
- Every block computes one tile of C .
- All threads in one block operate on submatrices:

$$C_{i,j} = \sum_{k=1}^{m/w} A_{i,k} \cdot B_{k,j}.$$

- The submatrices $A_{i,k}$ and $B_{k,j}$ are loaded from global memory into shared memory of the block.

matrix multiplication with shared memory

$$C_{i,j} = \sum_{k=1}^{m/w} A_{i,k} \cdot B_{k,j}$$



$\leftarrow m \rightarrow$

$\leftarrow p \rightarrow$

Device Memories and Matrix Multiplication

1

Device Memories

- global, constant, and shared memories
- CUDA variable type qualifiers

2

Matrix Multiplication

- an application of tiling
- **running `matrixMul` in the GPU Computing SDK**
- the kernel of `matrixMul`

the `matrixMul` in the GPU Computing SDK

The matrix-matrix multiplication is explained in great detail in the CUDA programming guide.

One of the examples in the GPU Computing SDK is `matrixMul`.

We run it on the GeForce 9400M, the Tesla C2050/C2070, the Tesla K20c (on `kepler`), and P100 (on `pascal`).

on the GeForce 9400M

```
/Developer/GPU Computing/C/bin/darwin/release $ ./matrixMul
[matrixMul] starting...

[ matrixMul ]
./matrixMul
Starting (CUDA and CUBLAS tests)...

Device 0: "GeForce 9400M" with Compute 1.1 capability

Using Matrix Sizes: A(160 x 320), B(160 x 320), C(160 x 320)

Runing Kernels...

> CUBLAS      7.2791 GFlop/s, Time = 0.00225 s, Size = 16384000 Ops

> CUDA matrixMul 5.4918 GFlop/s, Time = 0.00298 s, Size = 16384000 Ops, \
NumDevsUsed = 1, Workgroup = 256

Comparing GPU results with Host computation...

Comparing CUBLAS & Host results
CUBLAS compares OK

Comparing CUDA matrixMul & Host results
CUDA matrixMul compares OK

[matrixMul] test results...
PASSED
```

on the Tesla C2050/C2070

```
/usr/local/cuda/sdk/C/bin/linux/release jan$ ./matrixMul
[matrixMul] starting...
[ matrixMul ]
./matrixMul Starting (CUDA and CUBLAS tests)...

Device 0: "Tesla C2050 / C2070" with Compute 2.0 capability

Using Matrix Sizes: A(640 x 960), B(640 x 640), C(640 x 960)

Runing Kernels...

> CUBLAS      Throughput = 424.8840 GFlop/s, Time = 0.00185 s, \
Size = 786432000 Ops

> CUDA matrixMul Throughput = 186.7684 GFlop/s, Time = 0.00421 s, \
Size = 786432000 Ops, NumDevsUsed = 1, Workgroup = 1024

Comparing GPU results with Host computation...

Comparing CUBLAS & Host results
CUBLAS compares OK

Comparing CUDA matrixMul & Host results
CUDA matrixMul compares OK

[matrixMul] test results...
PASSED
```



on the K20c

```
$ /usr/local/cuda/samples/0_Simple/matrixMul/matrixMul
[Matrix Multiply Using CUDA] - Starting...

GPU Device 0: "Tesla K20c" with compute capability 3.5

MatrixA(320,320), MatrixB(640,320)
Computing result using CUDA Kernel...
done
Performance= 246.13 GFlop/s, Time= 0.533 msec, Size= 131072000 Ops,
WorkgroupSize= 1024 threads/block
Checking computed result for correctness: Result = PASS

Note: For peak performance, please refer to the matrixMulCUBLAS \
example.
$
```

The theoretical peak performance of the K20c is 1.17 TFlops double precision, and 3.52 TFlops single precision.

The matrices that are multiplied have single float as type.

going for peak performance with CUBLAS

```
$ /usr/local/cuda/samples/0_Simple/matrixMulCUBLAS/matrixMulCUBLAS  
[Matrix Multiply CUBLAS] - Starting...  
/usr/bin/nvidia-modprobe: unrecognized option: "-u"  
  
GPU Device 0: "Tesla K20c" with compute capability 3.5  
  
MatrixA(320,640), MatrixB(320,640), MatrixC(320,640)  
Computing result using CUBLAS...done.  
Performance= 1171.83 GFlop/s, Time= 0.112 msec, Size= 131072000 Ops  
Computing result using host CPU...done.  
Comparing CUBLAS Matrix Multiply with CPU results: PASS  
$
```

The theoretical peak performance of the K20c is 1.17 TFlops double precision, and 3.52 TFlops single precision.

The matrices that are multiplied have single float as type.

on the P100

```
$ /usr/local/cuda/samples/0_Simple/matrixMul/matrixMul
[Matrix Multiply Using CUDA] - Starting...
GPU Device 0: "Tesla P100-PCIE-16GB" with compute capability 6.0

MatrixA(320,320), MatrixB(640,320)
Computing result using CUDA Kernel...
done
Performance= 1909.26 GFlop/s, Time= 0.069 msec, Size= 131072000 Ops
WorkgroupSize= 1024 threads/block
Checking computed result for correctness: Result = PASS
```

NOTE: The CUDA Samples are not meant for performance measurements.
Results may vary when GPU Boost is enabled.

\$

The theoretical peak performance (with GPU Boost):
18.7 TFlops (half), 9.3 TFlops (single), 4.7 TFlops (double).

running CUBLAS on P100

```
$ /usr/local/cuda/samples/0_Simple/matrixMulCUBLAS/matrixMulCUBLAS  
[Matrix Multiply CUBLAS] - Starting...  
GPU Device 0: "Tesla P100-PCIE-16GB" with compute capability 6.0  
  
MatrixA(640,480), MatrixB(480,320), MatrixC(640,320)  
Computing result using CUBLAS...done.  
Performance= 3089.82 GFlop/s, Time= 0.064 msec, Size= 196608000 Ops  
Computing result using host CPU...done.  
Comparing CUBLAS Matrix Multiply with CPU results: PASS  
  
NOTE: The CUDA Samples are not meant for performance measurements.  
Results may vary when GPU Boost is enabled.  
$
```

A second run gave the following:

```
Performance= 3106.43 GFlop/s, Time= 0.063 msec, Size= 196608000 Ops
```

For single floats, the theoretical peak performance is 9.3 TFlops.

Device Memories and Matrix Multiplication

1

Device Memories

- global, constant, and shared memories
- CUDA variable type qualifiers

2

Matrix Multiplication

- an application of tiling
- running `matrixMul` in the GPU Computing SDK
- **the kernel of `matrixMul`**

the kernel of matrixMul

```
template <int BLOCK_SIZE> __global__ void
matrixMul( float* C, float* A, float* B, int wA, int wB)
{
    int bx = blockIdx.x;      // Block index
    int by = blockIdx.y;
    int tx = threadIdx.x;    // Thread index
    int ty = threadIdx.y;
    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;
    // Index of the last sub-matrix of A processed by the block
    int aEnd   = aBegin + wA - 1;
    // Step size used to iterate through the sub-matrices of A
    int aStep   = BLOCK_SIZE;
    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;
    // Step size used to iterate through the sub-matrices of B
    int bStep   = BLOCK_SIZE * wB;
```

the submatrices

```
// Csub is used to store the element of the block sub-matrix
// that is computed by the thread
float Csub = 0;

// Loop over all the sub-matrices of A and B
// required to compute the block sub-matrix
for (int a = aBegin, b = bBegin;
     a <= aEnd;
     a += aStep, b += bStep) {

    // Declaration of the shared memory array As used to
    // store the sub-matrix of A
    __shared__ float As[BLOCK_SIZE] [BLOCK_SIZE];

    // Declaration of the shared memory array Bs used to
    // store the sub-matrix of B
    __shared__ float Bs[BLOCK_SIZE] [BLOCK_SIZE];
```

loading and multiplying

```
// Load the matrices from device memory
// to shared memory; each thread loads
// one element of each matrix
AS(ty, tx) = A[a + wA * ty + tx];
BS(ty, tx) = B[b + wB * ty + tx];

// Synchronize to make sure the matrices are loaded
__syncthreads();

// Multiply the two matrices together;
// each thread computes one element
// of the block sub-matrix
#pragma unroll
for (int k = 0; k < BLOCK_SIZE; ++k)
    Csub += AS(ty, k) * BS(k, tx);

// Synchronize to make sure that the preceding
// computation is done before loading two new
// sub-matrices of A and B in the next iteration
__syncthreads();
}
```

the end of the kernel

```
// Write the block sub-matrix to device memory;  
// each thread writes one element  
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;  
C[c + wB * ty + tx] = Csub;  
}
```

The emphasis in this lecture is on

- ① the use of device memories; and
- ② data organization (tiling) and transfer.

In the next lecture we will come back to this code,
and cover thread scheduling

- ① the use of `blockIdx`; and
- ② thread synchronization.

summary and exercises

Vasily Volkov and James W. Demmel: **Benchmarking GPUs to tune dense linear algebra.** In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008. Article No. 31.

We covered more of chapter 3 in the book of Kirk & Hwu, and also several concepts explained in chapter 5.

- ① Compile the `matrixMul` of the GPU Computing SDK on your laptop and desktop and run the program.
- ② Consider the matrix multiplication code of last lecture and compute the CGMA ratio.
- ③ Adjust the code for matrix multiplication we discussed last time to use shared memory.

Thread Organization and Matrix Multiplication

1 Thread Organization

- grids, blocks, and threads
- using `threadIdx` and `blockIdx`
- setting the execution configuration parameters

2 Matrix Matrix Multiplication

- accessing submatrices with thread identifiers
- CUDA code for thread organization
- thread synchronization
- revisiting the kernel of `matrixMul`

MCS 572 Lecture 33
Introduction to Supercomputing
Jan Verschelde, 7 November 2016

Thread Organization and Matrix Multiplication

1 Thread Organization

- grids, blocks, and threads
- using `threadIdx` and `blockIdx`
- setting the execution configuration parameters

2 Matrix Matrix Multiplication

- accessing submatrices with thread identifiers
- CUDA code for thread organization
- thread synchronization
- revisiting the kernel of `matrixMul`

grids, blocks, and threads

The code that runs on the GPU is defined in a function, the kernel.

A kernel launch

- creates a grid of blocks, and
- each block has one or more threads.

The organization of the grids and blocks can be 1D, 2D, or 3D.

During the running of the kernel:

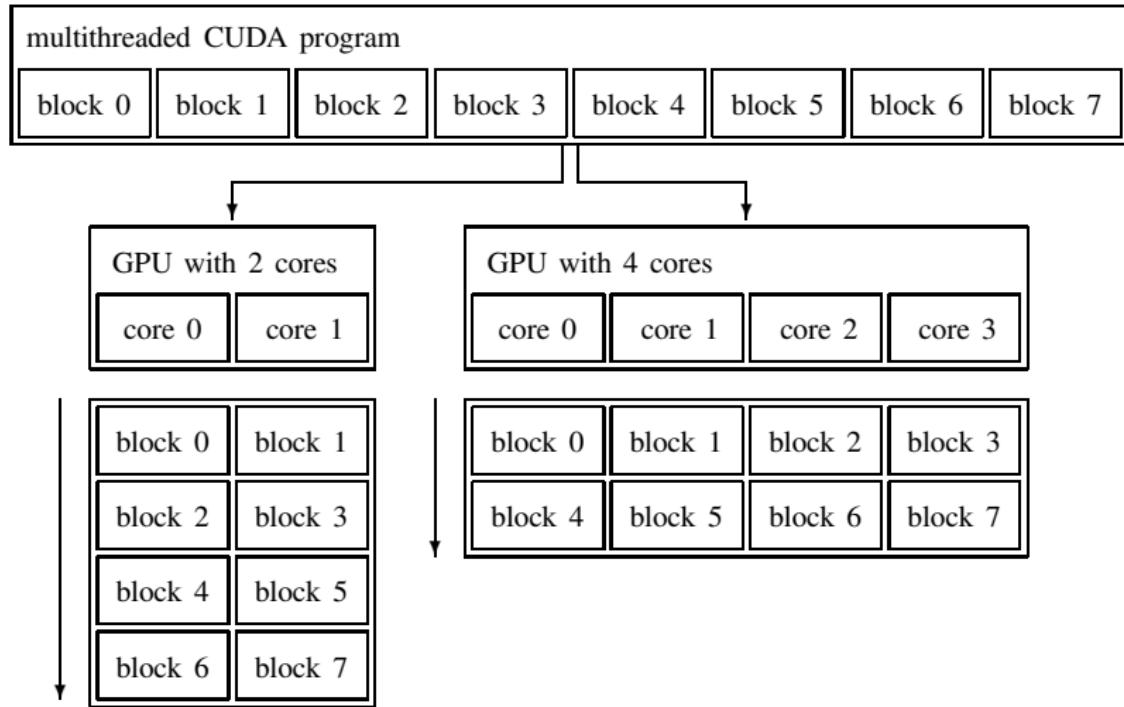
- Threads in the same block are executed simultaneously.
- Blocks are scheduled by the streaming multiprocessors.

The NVIDIA Tesla C2050 has 14 streaming multiprocessors and threads are executed in groups of 32 (the warp size).

This implies: $14 \times 32 = 448$ threads can run simultaneously.

For the K20c the numbers are respectively 13, 192, and 2496; and for the P100, we have 56, 64, and 3584.

a scalable programming model



Thread Organization and Matrix Multiplication

1

Thread Organization

- grids, blocks, and threads
- **using threadIdx and blockIdx**
- setting the execution configuration parameters

2

Matrix Matrix Multiplication

- accessing submatrices with thread identifiers
- CUDA code for thread organization
- thread synchronization
- revisiting the kernel of matrixMul

identifying threads

All threads execute the same code, defined by the kernel.

The builtin variable `threadIdx`

- identifies every thread in a block uniquely; and
- defines the data processed by the thread.

The builtin variable `blockDim` holds the number of threads in a block.

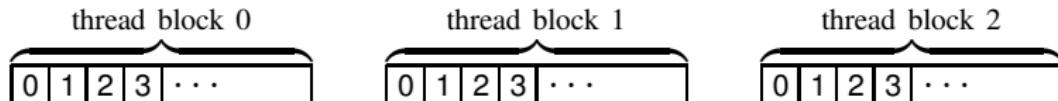
In a one dimensional organization, we use only `threadIdx.x` and `blockDim.x`. For 2D and 3D, the other components

- `threadIdx.y` belongs to the range `0..blockDim.y`;
- `threadIdx.z` belongs to the range `0..blockDim.z`.

data for each thread

The grid consists of N blocks, with $\text{blockIdx.x} \in \{0, N - 1\}$.

Within each block, $\text{threadIdx.x} \in \{0, \text{blockDim.x} - 1\}$.



```
int threadId = blockIdx.x *  
    blockDim.x + threadIdx.x  
...  
float x = input[threadID]  
float y = f(x)  
output[threadID] = y  
...
```

Thread Organization and Matrix Multiplication

1

Thread Organization

- grids, blocks, and threads
- using `threadIdx` and `blockIdx`
- setting the execution configuration parameters

2

Matrix Matrix Multiplication

- accessing submatrices with thread identifiers
- CUDA code for thread organization
- thread synchronization
- revisiting the kernel of `matrixMul`

setting the execution configuration parameters

Suppose the kernel is defined by the function `F` with input arguments `x` and output arguments `y`, then

```
dim3 dimGrid(128,1,1);  
dim3 dimBlock(32,1,1);  
F<<<dimGrid,dimBlock>>>(x,y);
```

launches a grid of 128 blocks. The grid is a one dimensional array. Each block in the grid is also one dimensional and has 32 threads.

multidimensional thread organization

Limitations of the Tesla C2050/C2070:

- Maximum number of threads per block: 1,024.
- Maximum sizes of each dimension of a block: $1,024 \times 1,024 \times 64$.
Because 1,024 is the upper limit for the number of threads in a block, the largest square 2D block is 32×32 , as $32^2 = 1,024$.
- Maximum sizes of each dimension of a grid:
 $65,535 \times 65,535 \times 65,535$.
 $65,535$ is the upper limit for the builtin variables `gridDim.x`, `gridDim.y`, and `gridDim.z`.

Limitations of the K20c and the P100:

- Maximum number of threads per block: 1,024.
- Maximum dimension size of a thread block: $1,024 \times 1,024 \times 64$.
- Maximum dimension size of a grid size:
 $2,147,483,647 \times 65,535 \times 65,535$

a 3D example

Suppose the function `F` defines the kernel,
with argument `x`, then

```
dim3 dimGrid(3,2,4);  
dim3 dimBlock(5,6,2);  
F<<<dimGrid, dimBlock>>>(x);
```

launches a grid with

- $3 \times 2 \times 4$ blocks; and
- each block has $5 \times 6 \times 2$ threads.

Thread Organization and Matrix Multiplication

1

Thread Organization

- grids, blocks, and threads
- using `threadIdx` and `blockIdx`
- setting the execution configuration parameters

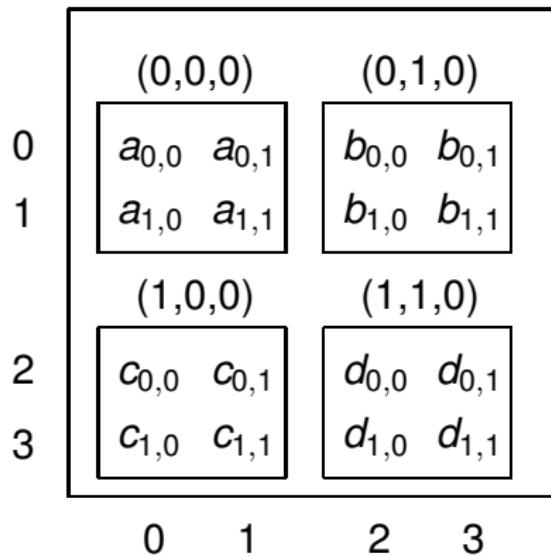
2

Matrix Matrix Multiplication

- accessing submatrices with thread identifiers
- CUDA code for thread organization
- thread synchronization
- revisiting the kernel of `matrixMul`

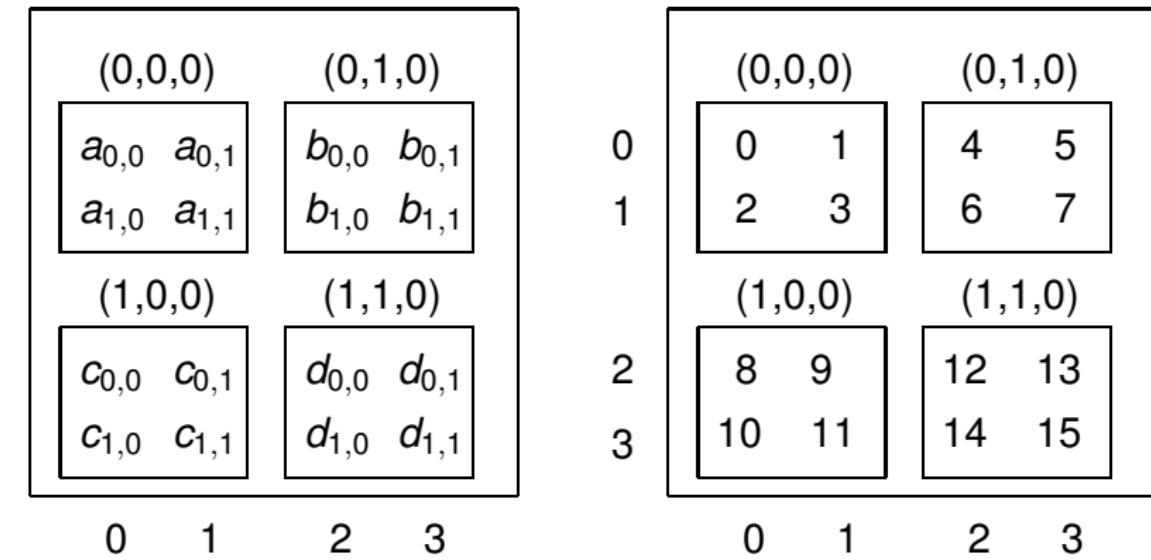
submatrices

Consider a grid of dimension $2 \times 2 \times 1$
to store a 4-by-4 matrix in tiles of dimensions $2 \times 2 \times 1$:



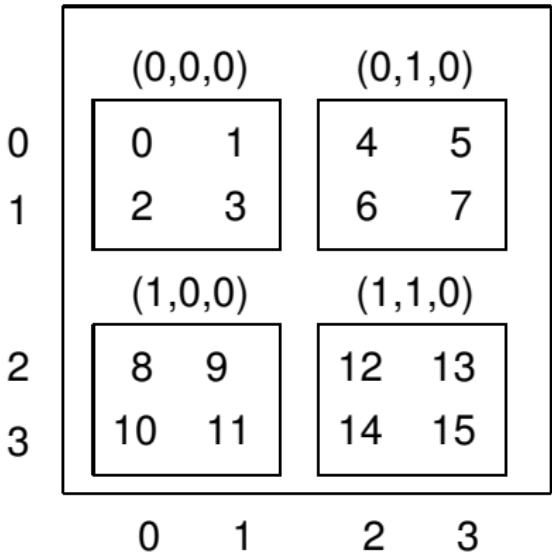
mapping threads to entries in the matrix

A kernel launch with a grid of dimensions $2 \times 2 \times 1$
where each block has dimensions $2 \times 2 \times 1$ creates 16 threads.



linear address calculation

A kernel launch with a grid of dimensions $2 \times 2 \times 1$
where each block has dimensions $2 \times 2 \times 1$ creates 16 threads.



| | | | | | | |
|------------|------------|------------|------------|------------|------------|------|
| x[0][0][0] | x[0][0][0] | x[0][0][0] | x[0][0][0] | x[0][0][0] | x[0][0][0] | = 0 |
| x[0][0][0] | x[0][0][0] | x[0][0][0] | x[0][0][1] | x[0][0][0] | x[0][0][0] | = 1 |
| x[0][0][0] | x[0][0][0] | x[0][0][1] | x[0][0][0] | x[0][0][0] | x[0][0][0] | = 2 |
| x[0][0][0] | x[0][0][0] | x[0][0][1] | x[0][1][1] | x[0][0][0] | x[0][0][0] | = 3 |
| x[0][0][1] | x[0][0][0] | x[0][0][0] | x[0][0][0] | x[0][0][0] | x[0][0][0] | = 4 |
| x[0][0][1] | x[0][0][0] | x[0][0][0] | x[0][0][0] | x[0][1][0] | x[0][0][0] | = 5 |
| x[0][0][1] | x[0][0][0] | x[0][0][1] | x[0][1][0] | x[0][0][0] | x[0][0][0] | = 6 |
| x[0][0][1] | x[0][0][0] | x[0][1][1] | x[0][1][0] | x[0][0][0] | x[0][0][0] | = 7 |
| x[1][0][0] | x[0][0][0] | x[0][0][0] | x[0][0][0] | x[0][0][0] | x[0][0][0] | = 8 |
| x[1][0][0] | x[0][0][0] | x[0][0][0] | x[0][0][1] | x[0][1][0] | x[0][0][0] | = 9 |
| x[1][0][0] | x[0][0][0] | x[0][0][1] | x[0][1][0] | x[0][0][0] | x[0][0][0] | = 10 |
| x[1][0][0] | x[0][0][0] | x[0][1][1] | x[0][1][0] | x[0][0][0] | x[0][0][0] | = 11 |
| x[1][0][1] | x[0][0][0] | x[0][0][0] | x[0][0][0] | x[0][0][0] | x[0][0][0] | = 12 |
| x[1][0][1] | x[0][0][0] | x[0][0][0] | x[0][0][1] | x[0][1][0] | x[0][0][0] | = 13 |
| x[1][1][0] | x[0][0][0] | x[0][0][1] | x[0][1][0] | x[0][0][0] | x[0][0][0] | = 14 |
| x[1][1][0] | x[0][0][1] | x[0][1][1] | x[0][1][0] | x[0][0][0] | x[0][0][0] | = 15 |

Thread Organization and Matrix Multiplication

1

Thread Organization

- grids, blocks, and threads
- using `threadIdx` and `blockIdx`
- setting the execution configuration parameters

2

Matrix Matrix Multiplication

- accessing submatrices with thread identifiers
- **CUDA code for thread organization**
- thread synchronization
- revisiting the kernel of `matrixMul`

the main program

```
int main ( int argc, char* argv[] )  
{  
    const int xb = 2; /* gridDim.x */  
    const int yb = 2; /* gridDim.y */  
    const int zb = 1; /* gridDim.z */  
    const int xt = 2; /* blockDim.x */  
    const int yt = 2; /* blockDim.y */  
    const int zt = 1; /* blockDim.z */  
    const int n = xb*yb*zb*xt*yt*zt;  
  
    printf("allocating array of length %d...\n", n);  
  
    /* allocating and initializing on the host */  
  
    int *xhost = (int*)calloc(n, sizeof(int));  
    for(int i=0; i<n; i++) xhost[i] = -1.0;
```

copy to device and kernel launch

```
int *xdevice;
size_t sx = n*sizeof(int);
cudaMalloc((void**)&xdevice,sx);
cudaMemcpy(xdevice,xhost,sx,cudaMemcpyHostToDevice);

/* set the execution configuration for the kernel */

dim3 dimGrid(xb,yb,zb);
dim3 dimBlock(xt,yt,zt);
matrixFill<<<dimGrid,dimBlock>>>(xdevice);
```

the kernel definition

```
__global__ void matrixFill ( int *x )
/*
 * Fills the matrix using blockIdx and threadIdx. */
{
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int row = by*blockDim.y + ty;
    int col = bx*blockDim.x + tx;
    int dim = gridDim.x*blockDim.x;
    int i = row*dim + col;
    x[i] = i;
}
```

copying to host and writing the result

```
/* copy data from device to host */
cudaMemcpy(xhost, xdevice, sx, cudaMemcpyDeviceToHost);
cudaFree(xdevice);

int *p = xhost;
for(int i1=0; i1 < xb; i1++)
    for(int i2=0; i2 < yb; i2++)
        for(int i3=0; i3 < zb; i3++)
            for(int i4=0; i4 < xt; i4++)
                for(int i5=0; i5 < yt; i5++)
                    for(int i6=0; i6 < zt; i6++)
                        printf("x[%d] [%d] [%d] [%d] [%d] [%d] = %d\n",
                               i1, i2, i3, i4, i5, i6, *(p++));
return 0;
}
```

Thread Organization and Matrix Multiplication

1

Thread Organization

- grids, blocks, and threads
- using `threadIdx` and `blockIdx`
- setting the execution configuration parameters

2

Matrix Matrix Multiplication

- accessing submatrices with thread identifiers
- CUDA code for thread organization
- **thread synchronization**
- revisiting the kernel of `matrixMul`

thread synchronization

In a block all threads run independently.

CUDA allows threads in the same block to coordinate their activities using a barrier synchronization function:

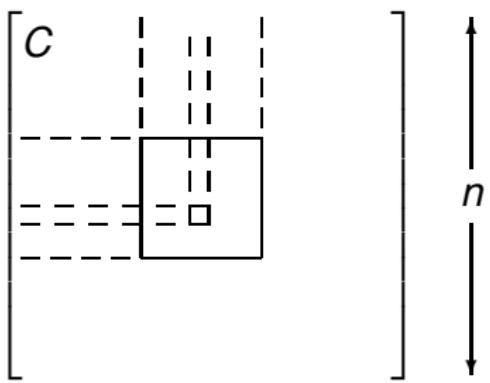
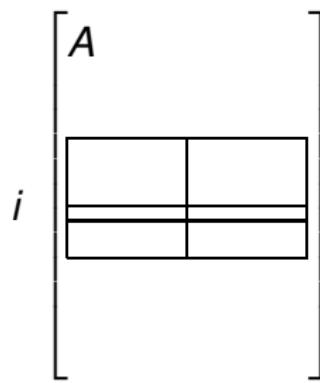
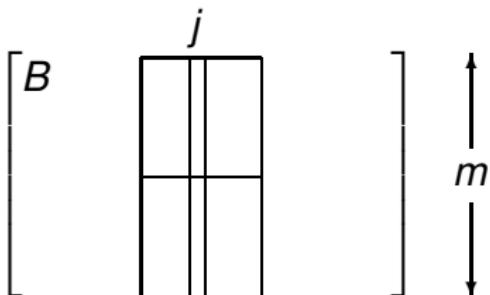
```
__syncthreads();
```

The thread executing `__syncthreads()` will be held at the calling location in the code until every thread in the block reaches the location.

Placing a `__syncthreads()` ensures that all threads in a block have completed a task before moving on.

applied to matrix multiplication with shared memory

$$C_{i,j} = \sum_{k=1}^{m/w} A_{i,k} \cdot B_{k,j}$$



application of `__syncthreads()`

With tiled matrix matrix multiplication using shared memory, all threads in the block collaborate to copy the tiles $A_{i,k}$ and $B_{k,j}$ from global memory to shared memory.

→ Before the calculation of the inner products, all threads must finish their copy statement: they all execute the `__syncthreads()`.

Every thread computes one inner product.

→ Before moving on to the next tile, all threads must finish, therefore, they all execute the `__syncthreads()` after computing their inner product and moving on to the next phase.

Thread Organization and Matrix Multiplication

1

Thread Organization

- grids, blocks, and threads
- using `threadIdx` and `blockIdx`
- setting the execution configuration parameters

2

Matrix Matrix Multiplication

- accessing submatrices with thread identifiers
- CUDA code for thread organization
- thread synchronization
- revisiting the kernel of `matrixMul`

the kernel of matrixMul

```
template <int BLOCK_SIZE> __global__ void
matrixMul( float* C, float* A, float* B, int wA, int wB)
{
    int bx = blockIdx.x;      // Block index
    int by = blockIdx.y;
    int tx = threadIdx.x;    // Thread index
    int ty = threadIdx.y;
    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;
    // Index of the last sub-matrix of A processed by the block
    int aEnd   = aBegin + wA - 1;
    // Step size used to iterate through the sub-matrices of A
    int aStep   = BLOCK_SIZE;
    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;
    // Step size used to iterate through the sub-matrices of B
    int bStep   = BLOCK_SIZE * wB;
```

the submatrices

```
// Csub is used to store the element of the block sub-matrix
// that is computed by the thread
float Csub = 0;

// Loop over all the sub-matrices of A and B
// required to compute the block sub-matrix
for (int a = aBegin, b = bBegin;
     a <= aEnd;
     a += aStep, b += bStep) {

    // Declaration of the shared memory array As used to
    // store the sub-matrix of A
    __shared__ float As[BLOCK_SIZE] [BLOCK_SIZE];

    // Declaration of the shared memory array Bs used to
    // store the sub-matrix of B
    __shared__ float Bs[BLOCK_SIZE] [BLOCK_SIZE];
```

loading and multiplying

```
// Load the matrices from device memory
// to shared memory; each thread loads
// one element of each matrix
AS(ty, tx) = A[a + wA * ty + tx];
BS(ty, tx) = B[b + wB * ty + tx];

// Synchronize to make sure the matrices are loaded
__syncthreads();

// Multiply the two matrices together;
// each thread computes one element
// of the block sub-matrix
#pragma unroll
for (int k = 0; k < BLOCK_SIZE; ++k)
    Csub += AS(ty, k) * BS(k, tx);

// Synchronize to make sure that the preceding
// computation is done before loading two new
// sub-matrices of A and B in the next iteration
__syncthreads();
}
```

the end of the kernel

```
// Write the block sub-matrix to device memory;  
// each thread writes one element  
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;  
C[c + wB * ty + tx] = Csub;  
}
```

Recommended reading:

- NVIDIA CUDA Programming Guide.
Available at <http://developer.nvidia.com>.
- Vasily Volkov and James W. Demmel: **Benchmarking GPUs to tune dense linear algebra.** In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008. Article No. 31.

summary and exercises

We covered more of chapter 4 in the book of Kirk & Hwu.

- ① Find the limitations of the grid and block sizes for the graphics card on your laptop or desktop.
- ② Extend the simple code with the three dimensional thread organization to a tiled matrix-vector multiplication for numbers generated at random as 0 or 1.

Warps and Reduction Algorithms

1 more on Thread Execution

- block partitioning into warps
- single-instruction, multiple-thread, and divergence

2 Parallel Reduction Algorithms

- computing the sum or the maximum
- a kernel for the parallel sum
- a kernel with less thread divergence

MCS 572 Lecture 34
Introduction to Supercomputing
Jan Verschelde, 9 November 2016

Warps and Reduction Algorithms

1 more on Thread Execution

- block partitioning into warps
- single-instruction, multiple-thread, and divergence

2 Parallel Reduction Algorithms

- computing the sum or the maximum
- a kernel for the parallel sum
- a kernel with less thread divergence

block partitioning into warps

The grid of threads are organized in a two level hierarchy:

- the grid is 1D, 2D, or 3D array of blocks; and
- each block is 1D, 2D, or 3D array of threads.

Blocks can execute in any order.

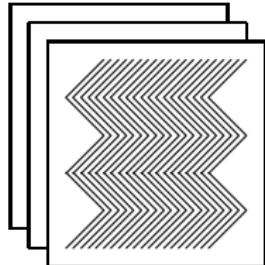
Threads are bundled for execution.

Each block is partitioned into *warps*.

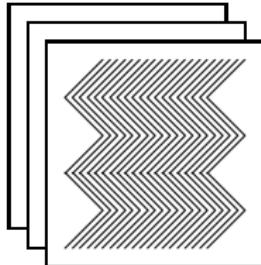
On the Tesla C2050/C2070, K20C, and P100,
each warp consists of 32 threads.

thread scheduling

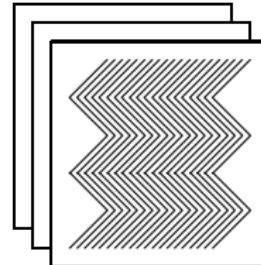
block 0 warps



block 1 warps



block 2 warps



streaming multiprocessor

instruction L1

instruction fetch/dispatch

shared memory

thread indices of warps

All threads in the same warp run at the same time.

The partitioning of threads in a one dimensional block,
for warps of size 32:

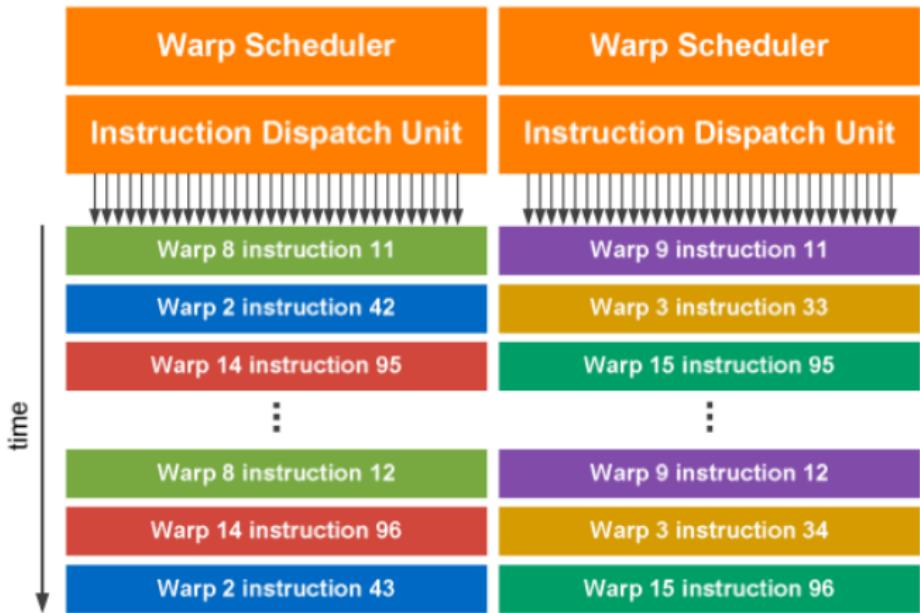
- warp 0 consists of threads 0 to 31 (value of `threadIdx`),
- warp w starts with thread $32w$ and ends with thread $32(w + 1) - 1$,
- the last warp is padded so it has 32 threads.

In a two dimensional block, threads in a warp are ordered along a lexicographical order of (`threadIdx.x, threadIdx.y`).

For example, an 8-by-8 block has 2 warps (of 32 threads):

- warp 0 has threads $(0, 0), (0, 1), \dots, (0, 7), (1, 0), (1, 1), \dots, (1, 7), (2, 0), (2, 1), \dots, (2, 7), (3, 0), (3, 1), \dots, (3, 7)$; and
- warp 1 has threads $(4, 0), (4, 1), \dots, (4, 7), (5, 0), (5, 1), \dots, (5, 7), (6, 0), (6, 1), \dots, (6, 7), (7, 0), (7, 1), \dots, (7, 7)$.

dual warp scheduler



from the NVIDIA Whitepaper

NVIDIA Next Generation CUDA Compute Architecture: Fermi.

why so many warps?

Why give so many warps to a streaming multiprocessor if there only 32 can run at the same time?

Answer: to efficiently execute long latency operations.

What about latency?

- A warp must often wait for the result of a global memory access
— — an example of a long latency operation — —
and is therefore not scheduled for execution.
- If another warp is ready for execution,
then that warp can be selected to execute the next instruction.

The mechanism of filling the latency of expensive operation with work from other threads is known as *latency hiding*.

zero overhead thread scheduling

Warp scheduling is used for other types of latency operations:

- pipelined floating point arithmetic,
- branch instructions.

With enough warps, the hardware will find a warp to execute, in spite of long latency operations.

The selection of ready warps for execution introduces no idle time and is referred to as *zero overhead thread scheduling*.

The long waiting time of warp instructions is hidden by executing instructions of other warps.

In contrast, CPUs tolerate latency operations with

- cache memories, and
- branch prediction mechanisms.

applied to matrix-matrix multiplication

For matrix-matrix multiplication,
what should the dimensions of the blocks of threads be?

We narrow the choices to three: 8×8 , 16×16 , or 32×32 ?

Considering that the C2050/C2070 has 14 streaming multiprocessors:

- ① $32 \times 32 = 1,024$ equals the limit of threads per block.
- ② $8 \times 8 = 64$ threads per block and $1,024/64 = 16$ blocks.
- ③ $16 \times 16 = 256$ threads per block and $1,024/256 = 4$ blocks.

Note that we must also take into account the size of shared memory
when executing tiled matrix matrix multiplication.

Warps and Reduction Algorithms

1 more on Thread Execution

- block partitioning into warps
- single-instruction, multiple-thread, and divergence

2 Parallel Reduction Algorithms

- computing the sum or the maximum
- a kernel for the parallel sum
- a kernel with less thread divergence

Single-Instruction, Multiple-Thread (SIMT)

In multicore CPUs, we use Single-Instruction, Multiple-Data (SIMD): the multiple data elements to be processed by a single instruction must be first collected and packed into a single register.

In SIMT, all threads process data in their own registers.

In SIMT, the hardware executes an instruction for all threads in the same warp, before moving to the next instruction.

This style of execution is motivated by hardware costs constraints.

The cost of fetching and processing an instruction is amortized over a large number of threads.

paths of execution

Single-Instruction, Multiple-Thread works well when all threads within a warp follow the same control flow path.

For example, for an *if-then-else* construct, it works well

- when either all threads execute the *then* part,
- or all execute the *else* part.

If threads within a warp take different control flow paths, then the SIMT execution style no longer works well.

thread divergence

Considering the *if-then-else* example, it may happen that

- some threads in a warp execute the *then* part,
- other threads in the same warp execute the *else* part.

In the SIMT execution style, multiple passes are required:

- one pass for the *then* part of the code, and
- another pass for the *else* part.

These passes are sequential to each other
and thus increase the execution time.

If threads in the same warp follow different paths of control flow,
then we say that these threads *diverge* in their execution.

other examples of thread divergence

Consider an iterative algorithm with a loop

- some threads finish in 6 iterations,
- other threads need 7 iterations.

In this example, two passes are required:

- ① one pass for those threads that do the 7th iteration,
- ② another pass for those threads that do not.

In some code, decisions are made on the `threadIdx` values:

- For example: `if(threadIdx.x > 2) { ... }`.
- The loop condition may be based on `threadIdx`.

An important class where thread divergence is likely to occur is the class of reduction algorithms.

Warps and Reduction Algorithms

1 more on Thread Execution

- block partitioning into warps
- single-instruction, multiple-thread, and divergence

2 Parallel Reduction Algorithms

- computing the sum or the maximum
- a kernel for the parallel sum
- a kernel with less thread divergence

reduction algorithms

A reduction algorithm extracts one value from an array, e.g.:

- the sum of an array of elements,
- the maximum or minimum element in an array.

A reduction algorithm visits every element in the array, using a current value for the sum or the maximum/minimum.

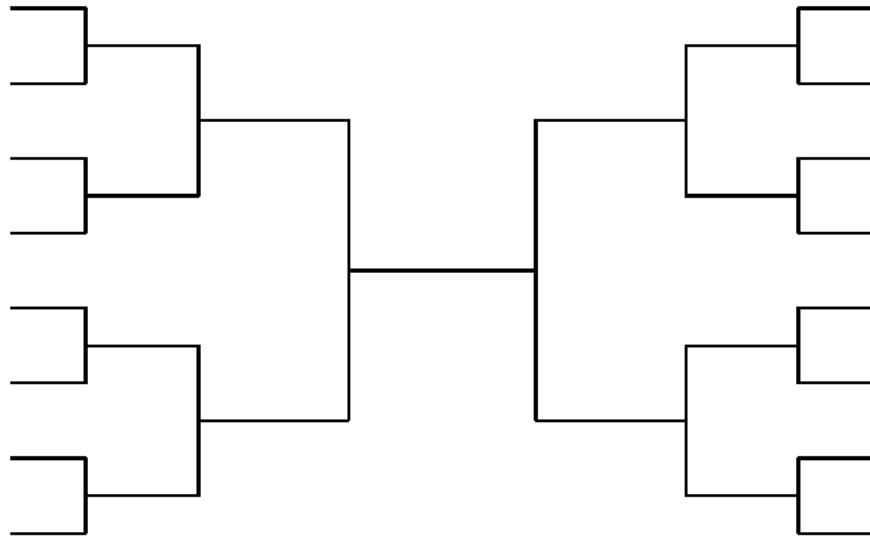
Large enough arrays motivate parallel execution of the reduction.

To reduce n elements, $n/2$ threads take $\log_2(n)$ steps.

Reduction algorithms take only 1 flop per element loaded:

- not *compute bound*, that is: limited by flops performance,
- but *memory bound*, that is: limited by memory bandwidth.

a tournament



Warps and Reduction Algorithms

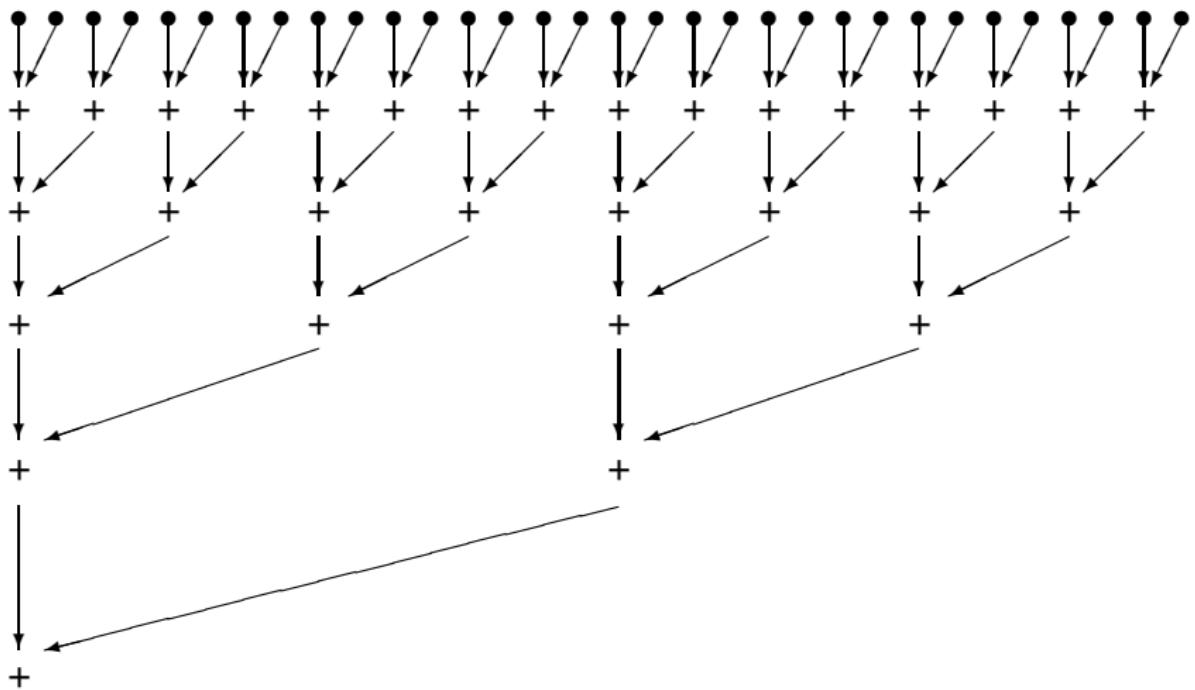
1 more on Thread Execution

- block partitioning into warps
- single-instruction, multiple-thread, and divergence

2 Parallel Reduction Algorithms

- computing the sum or the maximum
- **a kernel for the parallel sum**
- a kernel with less thread divergence

summing 32 numbers



code in a kernel to sum numbers

The original array is in the global memory
and copied to shared memory for a thread block to sum.

```
__shared__ float partialSum[];  
  
int t = threadIdx.x;  
for(int stride = 1; stride < blockDim.x; stride *= 2)  
{  
    __syncthreads();  
    if(t % (2*stride) == 0)  
        partialSum[t] += partialSum[t+stride];  
}
```

The reduction is done *in place*, replacing elements.

The `__syncthreads()` ensures that all partial sums from the previous iteration have been computed.

thread divergence in the first kernel

Because of the statement

```
if(t % (2*stride) == 0)
    partialSum[t] += partialSum[t+stride];
```

the kernel clearly has thread divergence.

In each iteration, two passes are needed to execute all threads, even though fewer threads will perform an addition.

Warps and Reduction Algorithms

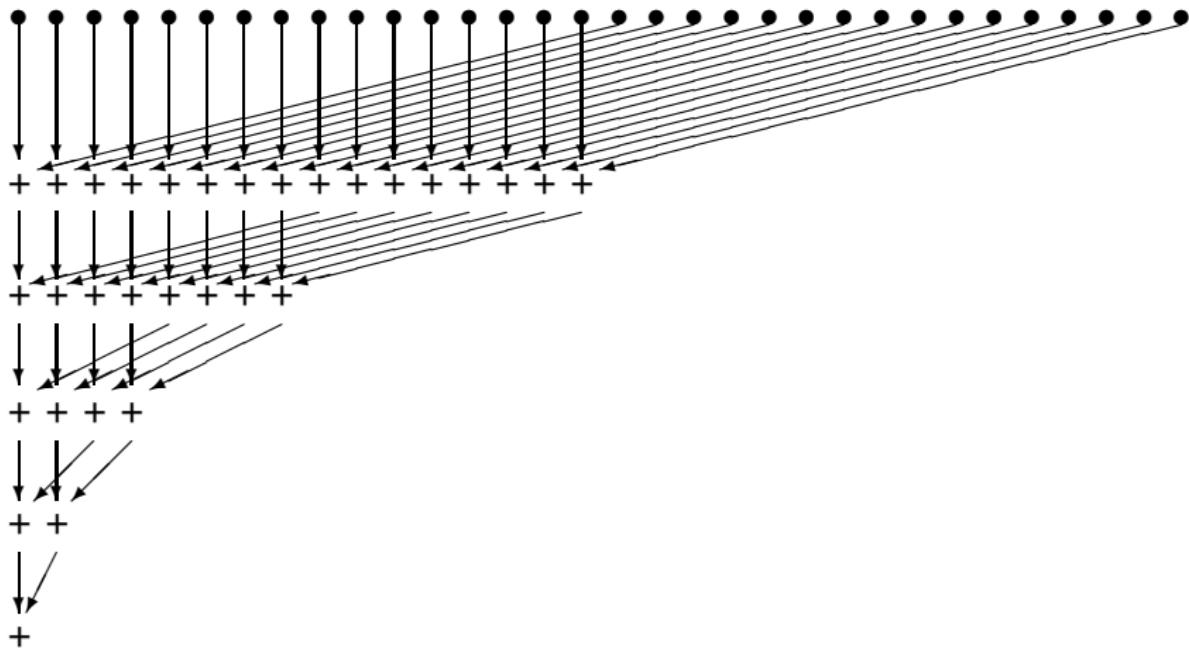
1 more on Thread Execution

- block partitioning into warps
- single-instruction, multiple-thread, and divergence

2 Parallel Reduction Algorithms

- computing the sum or the maximum
- a kernel for the parallel sum
- a kernel with less thread divergence

a different summation algorithm



the revised kernel

The original array is in the global memory
and copied to shared memory for a thread block to sum.

```
__shared__ float partialSum[];  
  
int t = threadIdx.x;  
for(int stride = blockDim.x >> 1; stride > 0;  
    stride >> 1)  
{  
    __syncthreads();  
    if(t < stride)  
        partialSum[t] += partialSum[t+stride];  
}
```

The division by 2 is done by shifting the stride value to the right by 1 bit.

why less thread divergence?

At first, there seems no improvement, because of the `if`.

Consider a block of 1,024 threads, partitioned in 32 warps.

A warp consists of 32 threads with consecutive `threadIdx` values:

- all threads in warp 0 to 15 execute the add statement,
- all threads in warp 16 to 31 skip the add statement.

All threads in each warp take the same path \Rightarrow no thread divergence.

If the number of threads that execute the add drops below 32, then thread divergence still occurs.

Thread divergence occurs in the last 5 iterations.

bibliography

- S. Sengupta, M. Harris, and M. Garland.
Efficient parallel scan algorithms for GPUs.
Technical Report NVR-2008-003, NVIDIA, 2008.
- M. Harris. **Optimizing parallel reduction in CUDA.**
White paper available at <http://docs.nvidia.com>.

summary and exercises

We covered §4.5 and §6.1 in the book of Kirk & Hwu, and discussed warp scheduling, latency hiding, SIMD, and thread divergence.
To illustrate the concepts we discussed two reduction algorithms.

- ① Consider the code `matrixMul` of the GPU computing SDK.
Look up the dimensions of the grid and blocks of threads.
Can you (experimentally) justify the choices made?
- ② Write code for the two summation algorithms we discussed.
Do experiments to see which algorithm performs better.
- ③ Apply the summation algorithm to the composite trapezoidal rule.

Use it to estimate π via
$$\frac{\pi}{4} = \int_0^1 \sqrt{1 - x^2} dx.$$

Memory Coalescing Techniques

1 Accessing Global and Shared Memory

- memory coalescing to global memory
- avoiding bank conflicts in shared memory

2 Memory Coalescing Techniques

- accessing global memory for a matrix
- using shared memory for coalescing

3 Avoiding Bank Conflicts

- computing consecutive powers

MCS 572 Lecture 35
Introduction to Supercomputing
Jan Verschelde, 11 November 2016

Memory Coalescing Techniques

1 Accessing Global and Shared Memory

- memory coalescing to global memory
- avoiding bank conflicts in shared memory

2 Memory Coalescing Techniques

- accessing global memory for a matrix
- using shared memory for coalescing

3 Avoiding Bank Conflicts

- computing consecutive powers

dynamic random access memories (DRAMs)

Accessing data in the global memory is critical to the performance of a CUDA application.

In addition to tiling techniques utilizing shared memories we discuss memory coalescing techniques to move data efficiently from global memory into shared memory and registers.

Global memory is implemented with dynamic random access memories (DRAMs). Reading one DRAM is a very slow process.

Modern DRAMs use a parallel process:

Each time a location is accessed, many consecutive locations that includes the requested location are accessed.

If an application uses data from consecutive locations before moving on to other locations, the DRAMs work close to the advertised peak global memory bandwidth.

memory coalescing

Recall that all threads in a warp execute the same instruction.

When all threads in a warp execute a load instruction, the hardware detects whether the threads access consecutive memory locations.

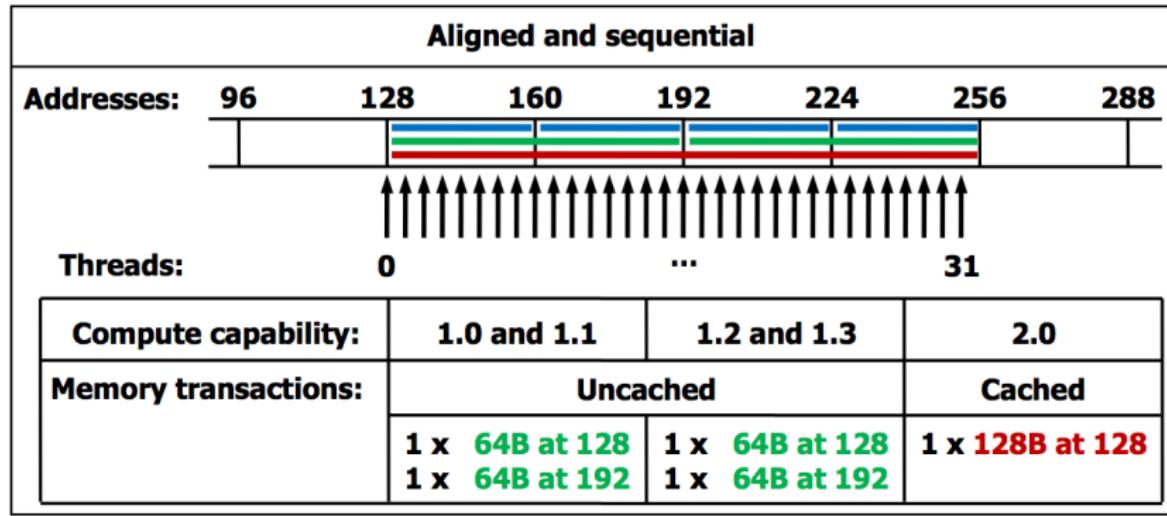
The most favorable global memory access is achieved when the same instruction for all threads in a warp accesses global memory locations.

In this favorable case, the hardware *coalesces* all memory accesses into a consolidated access to consecutive DRAM locations.

If thread 0 accesses location n , thread 1 accesses location $n + 1$, ... thread 31 accesses location $n + 31$, then all these accesses are *coalesced*, that is: combined into one single access.

The CUDA C Best Practices Guide gives a high priority recommendation to coalesced access to global memory.

an example of a global memory access by a warp



from Figure G-1 of the *NVIDIA Programming Guide*.

aligned memory access for higher compute capability

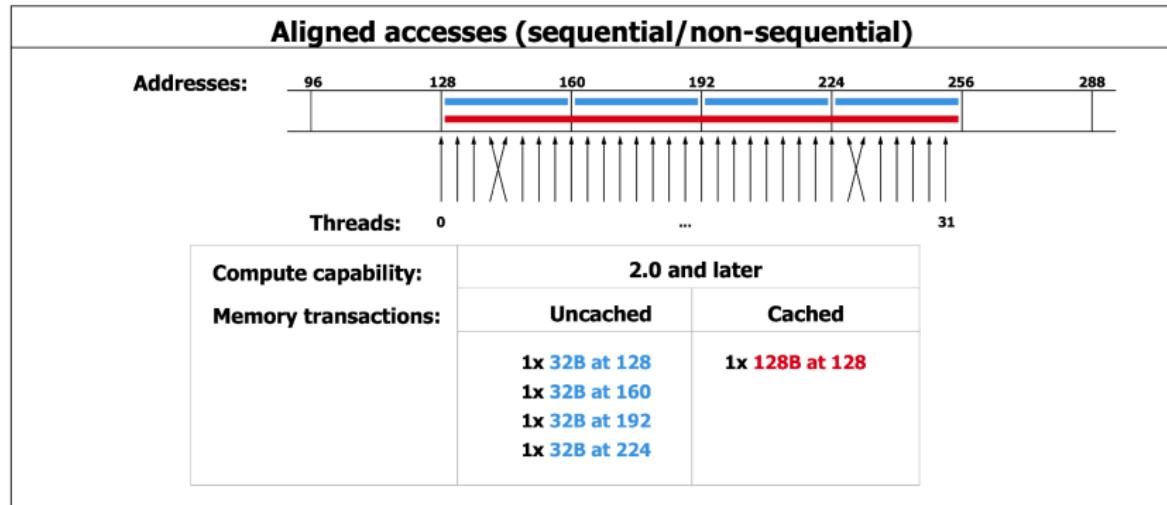


Figure 16 of the 2016 NVIDIA Programming Guide

mis-aligned memory access

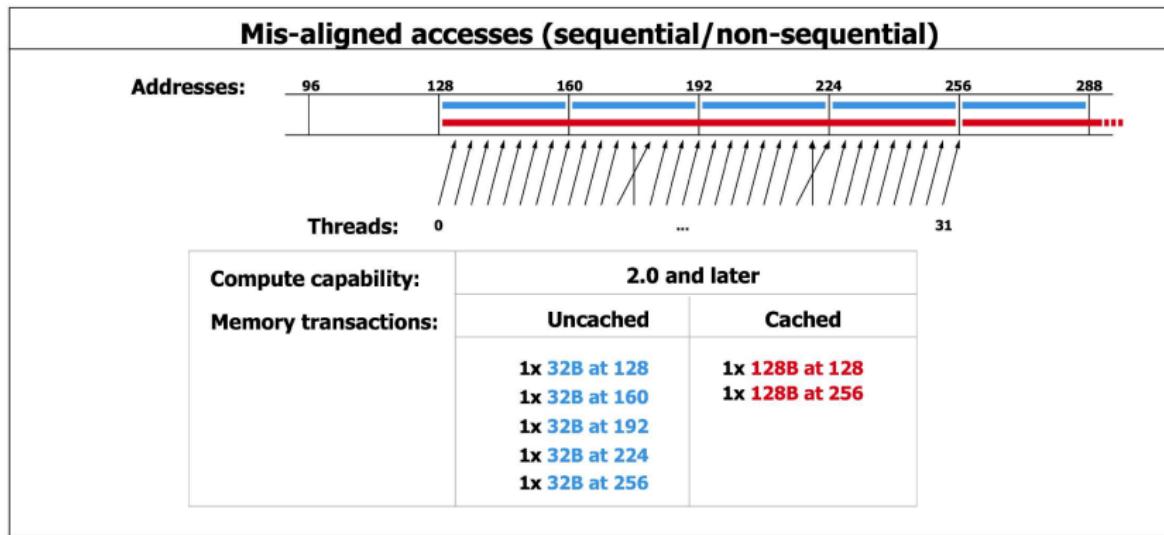


Figure 16 of the 2016 NVIDIA Programming Guide

alignment in memory

In /usr/local/cuda/include/vector_types.h
we find the definition of the type double2 as

```
struct __device_builtin__ __builtin_align__(16) double2
{
    double x, y;
};
```

The __align__(16) causes the doubles in double2 to be 16-byte or 128-bit aligned.

Using the double2 type for the real and imaginary part of a complex number allows for coalesced memory access.

exploring the effects of misaligned memory access

With a simple copy kernel we can explore what happens when access to global memory is misaligned:

```
__global__ void copyKernel
( float *output, float *input, int offset )
{
    int i = blockIdx.x*blockDim.x + threadIdx.x + offset;
    output[i] = input[i];
}
```

The bandwidth will decrease significantly for $offset > 1$.

Memory Coalescing Techniques

1 Accessing Global and Shared Memory

- memory coalescing to global memory
- avoiding bank conflicts in shared memory

2 Memory Coalescing Techniques

- accessing global memory for a matrix
- using shared memory for coalescing

3 Avoiding Bank Conflicts

- computing consecutive powers

shared memory and memory banks

Shared memory has 32 banks that are organized such that successive 32-bit words are assigned to successive banks, i.e.: interleaved.

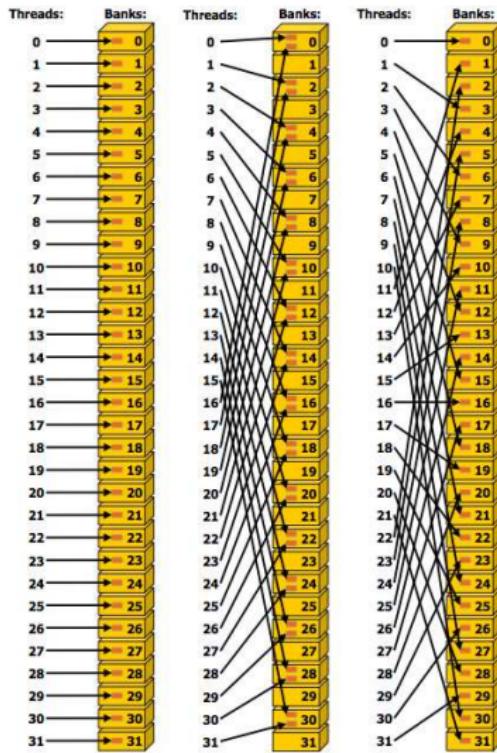
The bandwidth of shared memory is 32 bits per bank per clock cycle. Because shared memory is on chip, uncached shared memory latency is roughly $100\times$ lower than global memory.

A *bank conflict* occurs if two or more threads access any bytes within *different* 32-bit words belonging to the *same* bank.

If two or more threads access any bytes within the same 32-bit word, then there is no bank conflict between these threads.

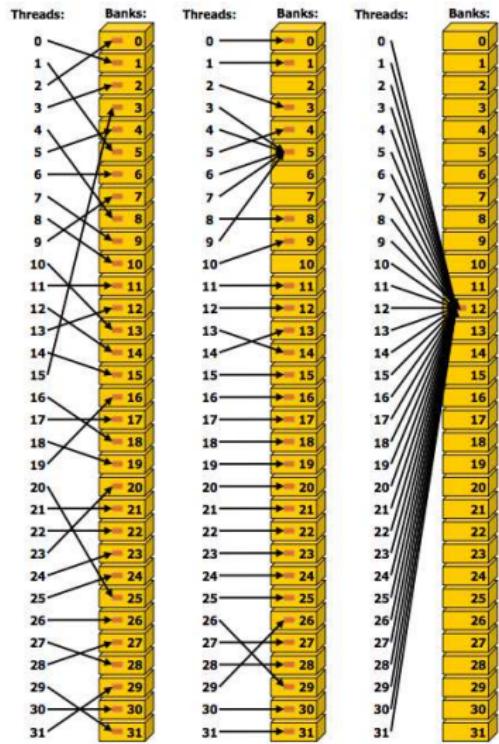
The CUDA C Best Practices Guide gives a medium priority recommendation to shared memory access without bank conflicts.

examples of strided shared memory accesses



from Figure G-2 of the *NVIDIA Programming Guide*.

irregular and colliding shared memory accesses



from Figure G-3 of the *NVIDIA Programming Guide*.

Memory Coalescing Techniques

1 Accessing Global and Shared Memory

- memory coalescing to global memory
- avoiding bank conflicts in shared memory

2 Memory Coalescing Techniques

- accessing global memory for a matrix
- using shared memory for coalescing

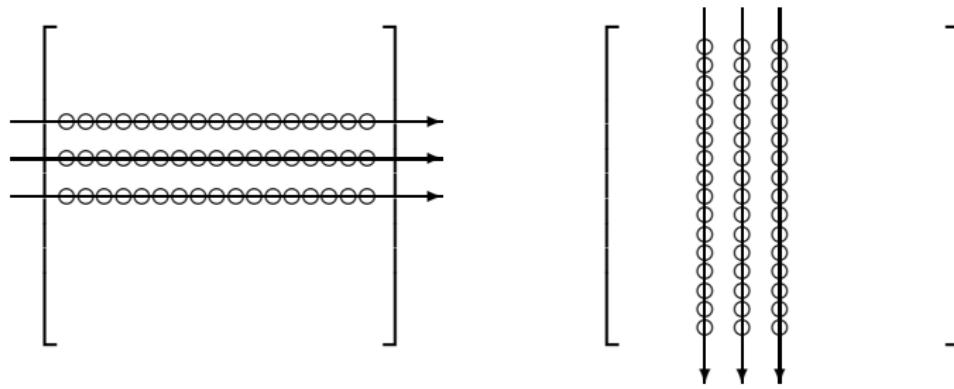
3 Avoiding Bank Conflicts

- computing consecutive powers

accessing the elements in a matrix

Consider two ways of accessing the elements in a matrix:

- ① elements are accessed row after row; or
- ② elements are accessed column after column.



linear address system

Consider a 4-by-4 matrix:

| | | | |
|-----------|-----------|-----------|-----------|
| $a_{0,0}$ | $a_{0,1}$ | $a_{0,2}$ | $a_{0,3}$ |
| $a_{1,0}$ | $a_{1,1}$ | $a_{1,2}$ | $a_{1,3}$ |
| $a_{2,0}$ | $a_{2,1}$ | $a_{2,2}$ | $a_{2,3}$ |
| $a_{3,0}$ | $a_{3,1}$ | $a_{3,2}$ | $a_{3,3}$ |

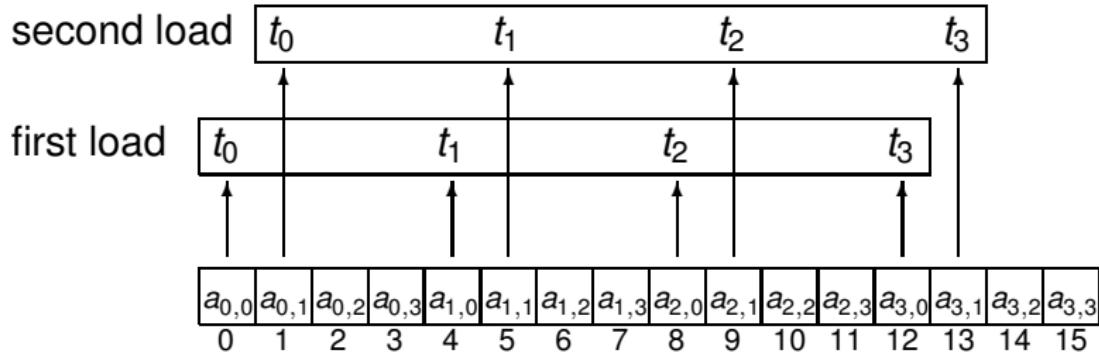
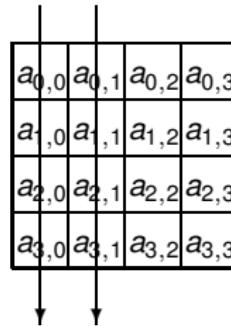


| | | | | | | | | | | | | | | | |
|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| $a_{0,0}$ | $a_{0,1}$ | $a_{0,2}$ | $a_{0,3}$ | $a_{1,0}$ | $a_{1,1}$ | $a_{1,2}$ | $a_{1,3}$ | $a_{2,0}$ | $a_{2,1}$ | $a_{2,2}$ | $a_{2,3}$ | $a_{3,0}$ | $a_{3,1}$ | $a_{3,2}$ | $a_{3,3}$ |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

In C, the matrix is stored row wise as a one dimensional array.

first access

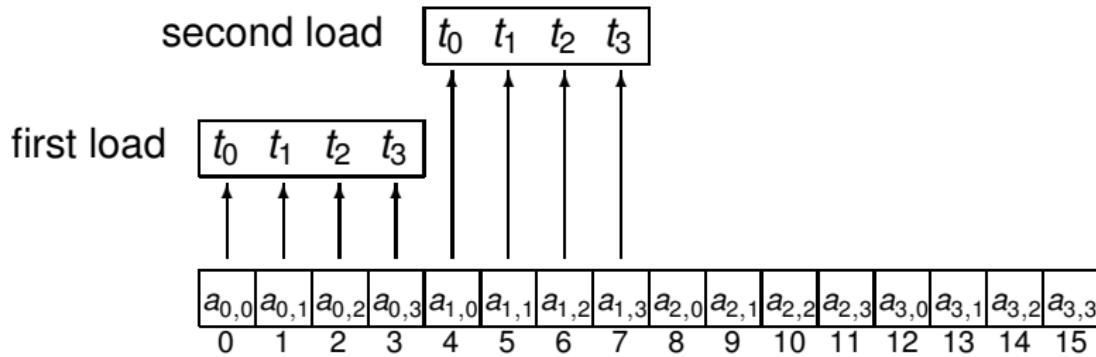
Threads t_0, t_1, t_2 , and t_3 access the elements on the first two columns:



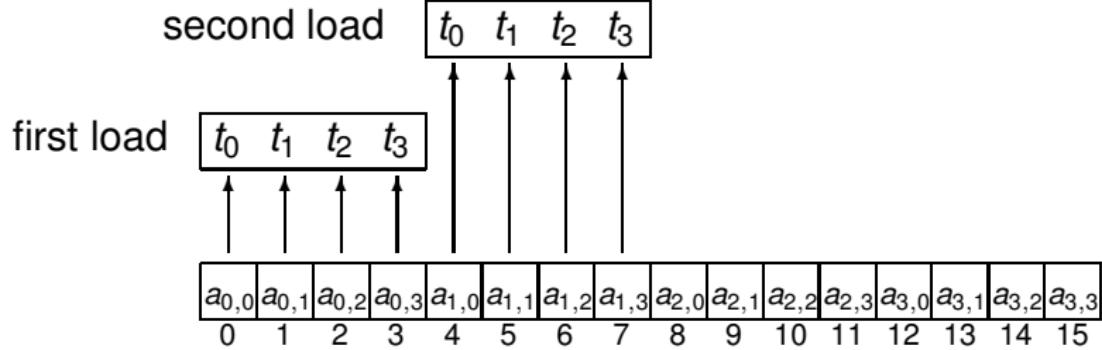
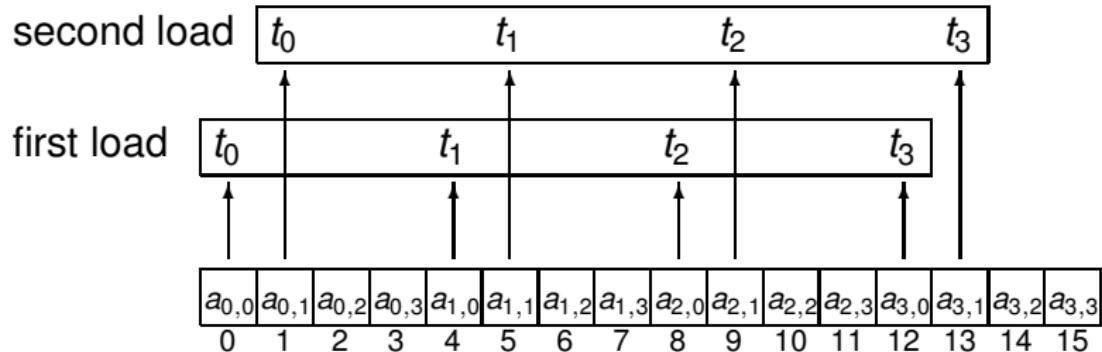
second access

Four threads t_0, t_1, t_2 , and t_3 access elements on the first two rows:

| | | | |
|-----------|-----------|-----------|-----------|
| $a_{0,0}$ | $a_{0,1}$ | $a_{0,2}$ | $a_{0,3}$ |
| $a_{1,0}$ | $a_{1,1}$ | $a_{1,2}$ | $a_{1,3}$ |
| $a_{2,0}$ | $a_{2,1}$ | $a_{2,2}$ | $a_{2,3}$ |
| $a_{3,0}$ | $a_{3,1}$ | $a_{3,2}$ | $a_{3,3}$ |



uncoalesced versus coalesced access



Memory Coalescing Techniques

1 Accessing Global and Shared Memory

- memory coalescing to global memory
- avoiding bank conflicts in shared memory

2 Memory Coalescing Techniques

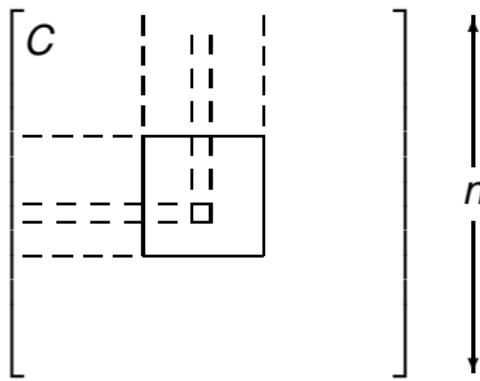
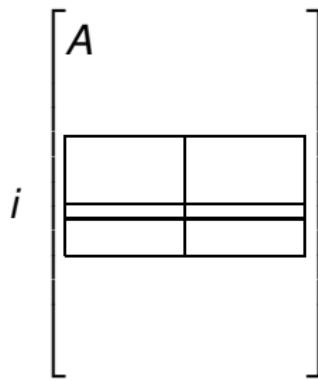
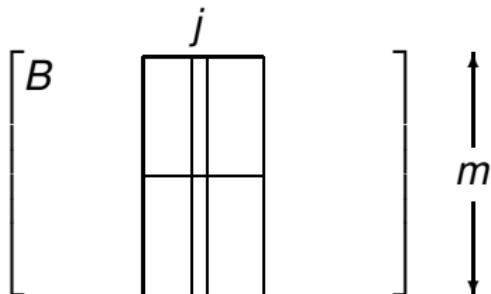
- accessing global memory for a matrix
- using shared memory for coalescing

3 Avoiding Bank Conflicts

- computing consecutive powers

tiled matrix-matrix multiplication

$$C_{i,j} = \sum_{k=1}^{m/w} A_{i,k} \cdot B_{k,j}$$



$\leftarrow m \rightarrow$

$\leftarrow p \rightarrow$

tiled matrix multiplication with shared memory

For $C_{i,j} = \sum_{k=1}^{m/w} A_{i,k} \cdot B_{k,j}$, $A \in \mathbb{R}^{n \times m}$, $B \in \mathbb{R}^{m \times p}$, $A_{i,k}, B_{k,j}, C_{i,j} \in \mathbb{R}^{w \times w}$,

every warp reads one tile $A_{i,k}$ of A and one tile $B_{k,j}$ of B : every thread in the warp reads one element of $A_{i,k}$ and one element of $B_{k,j}$.

The number of threads equals w , the width of one tile, and threads are identified with $tx = \text{threadIdx.x}$ and $ty = \text{threadIdx.y}$.

The $by = \text{blockIdx.y}$ and $bx = \text{blockIdx.x}$ correspond respectively to the first and the second index of each tile, so we have $\text{row} = by * w + ty$ and $\text{col} = bx * w + tx$.

Row wise access to A uses $A[\text{row} * m + (k * w + tx)]$. For B : $B[(k * w + ty) * m + \text{col}] = B[(k * w + ty) * m + bx * w + tx]$.

Adjacent threads in a warp have adjacent tx values so we have coalesced access also to B .

tiled matrix multiplication kernel

```
__global__ void mul ( float *A, float *B, float *C, int m )
{
    __shared__ float As[w][w];
    __shared__ float Bs[w][w];
    int bx = blockIdx.x;           int by = blockIdx.y;
    int tx = threadIdx.x;         int ty = threadIdx.y;
    int col = bx*w + tx;         int row = by*w + ty;
    float Cv = 0.0;
    for(int k=0; k<m/w; k++)
    {
        As[ty][tx] = A[row*m + (k*w + tx)];
        Bs[ty][tx] = B[(k*w + ty)*m + col];
        __syncthreads();
        for(int ell=0; ell<w; ell++)
            Cv += As[ty][ell]*Bs[ell][tx];
        C[row][col] = Cv;
    }
}
```

Memory Coalescing Techniques

1 Accessing Global and Shared Memory

- memory coalescing to global memory
- avoiding bank conflicts in shared memory

2 Memory Coalescing Techniques

- accessing global memory for a matrix
- using shared memory for coalescing

3 Avoiding Bank Conflicts

- computing consecutive powers

consecutive powers

Consider the following problem:

Input : $x_0, x_1, x_2, \dots, x_{31}$, all of type float.

Output : $x_0^2, x_0^3, x_0^4, \dots, x_0^{33}, x_1^2, x_1^3, x_1^4, \dots, x_1^{33}, x_2^2, x_2^3, x_2^4, \dots, x_2^{33},$
 $\dots, x_{31}^2, x_{31}^3, x_{31}^4, \dots, x_{31}^{33}.$

This gives 32 threads in a warp 1,024 multiplications to do.

Assume the input and output resides in shared memory.

How to compute without bank conflicts?

writing with stride

Observe the order of the output sequence:

Input : $x_0, x_1, x_2, \dots, x_{31}$, all of type float.

Output : $x_0^2, x_0^3, x_0^4, \dots, x_0^{33}, x_1^2, x_1^3, x_1^4, \dots, x_1^{33}, x_2^2, x_2^3, x_2^4, \dots, x_2^{33},$
 $\dots, x_{31}^2, x_{31}^3, x_{31}^4, \dots, x_{31}^{33}.$

If thread i computes $x_i^2, x_i^3, x_i^4, \dots, x_i^{33}$, then after the first step,
all threads write $x_0^2, x_1^2, x_2^2, \dots, x_{31}^2$ to shared memory.

If the stride is 32, all threads write into the same bank.

Instead of a simultaneous computation of 32 powers at once,
the writing to shared memory will be serialized.

changed order of storage

If we alter the order in the output sequence:

Input : $x_0, x_1, x_2, \dots, x_{31}$, all of type float.

Output : $x_0^2, x_1^2, x_2^2, \dots, x_{31}^2, x_0^3, x_1^3, x_2^3, \dots, x_{31}^3, x_0^4, x_1^4, x_2^4, \dots, x_{31}^4,$
 $\dots, x_0^{33}, x_1^{33}, x_2^{33}, \dots, x_{31}^{33}$.

After the first step, thread i writes x_i^2 in adjacent memory,
next to x_{i-1}^2 (if $i > 0$) and x_{i+1}^2 (if $i < 31$).

Without bank conflicts, the speedup will be close to 32.

summary and exercises

We covered §6.2 in the book of Kirk & Hwu.

- 1 Run `copyKernel` for large enough arrays for zero `offset` and an `offset` equal to two. Measure the timings and deduce the differences in memory bandwidth between the two different values for `offset`.
- 2 Consider the kernel of `matrixMul` in the GPU computing SDK. Is the loading of the tiles into shared memory coalesced? Justify your answer.
- 3 Write a CUDA program for the computation of consecutive powers, using coalesced access of the values for the input elements. Compare the two orders of storing the output sequence in shared memory: once with and once without bank conflicts.

Performance Considerations

1 Dynamic Partitioning of Resources

- streaming multiprocessor resources
- the CUDA occupancy calculator

2 the Compute Visual Profiler

- getting started with `computeprof`
- analysis of the kernel `matrixMul`

3 Data Prefetching and Instruction Mix

- registers between global and shared memory
- maximizing instruction throughput

MCS 572 Lecture 36
Introduction to Supercomputing
Jan Verschelde, 14 November 2016

Performance Considerations

1 Dynamic Partitioning of Resources

- streaming multiprocessor resources
- the CUDA occupancy calculator

2 the Compute Visual Profiler

- getting started with `computeprof`
- analysis of the kernel `matrixMul`

3 Data Prefetching and Instruction Mix

- registers between global and shared memory
- maximizing instruction throughput

streaming multiprocessor resources – part I

Comparing GPUs with respective compute capabilities 1.1, 2.0, 3.5, and 6.0: GeForce 9400M, Tesla C2050/C2070, K20C, P100:

| compute capability | 1.1 | 2.0 | 3.5 | 6.0 |
|---|-----|-------|-------|-----|
| maximum number of threads per block | 512 | 1,024 | | |
| maximum number of resident blocks per streaming multiprocessor | 8 | 16 | 32 | |
| warp size | | 32 | | |
| maximum number of resident warps per streaming multiprocessor | 24 | 48 | 64 | |
| maximum number of resident threads per streaming multiprocessor | 768 | 1,536 | 2,048 | |

data in the table from the CUDA C Programming Guide appendix G

dynamic partitioning of thread slots

During runtime, thread slots are partitioned and assigned to thread blocks.

Streaming multiprocessors are versatile by their ability to dynamically partition the thread slots among thread blocks.

They can

- either execute many thread blocks of few threads each,
- or execute a few thread blocks of many threads each.

In contrast, fixed partitioning where the number of blocks and threads per block are fixed will lead to waste.

Goal: keep multiprocessors fully occupied.

interactions between resource limitations – C2050

The Tesla C2050/C2070 has 1,536 thread slots per streaming multiprocessor. As $1,536 = 32 \times 48$, we have

$$\text{number of thread slots} = \text{warp size} \times \text{number of warps per block.}$$

For 32 threads per block, we have $1,536/32 = 48$ blocks

\leftrightarrow at most 8 blocks per streaming multiprocessor.

To fully utilize both the block and thread slots, to have 8 blocks, we should have

- $1,536/8 = 192$ threads per block, or
- $192/32 = 6$ warps per block.

interactions between resource limitations – K20C

The K20C has 2,048 thread slots per streaming multiprocessor.
As $2,048 = 32 \times 64$, we have

$$\text{number of thread slots} = \text{warp size} \times \text{number of warps per block.}$$

For 32 threads per block, we have $2,048/32 = 64$ blocks
 \leftrightarrow at most 16 blocks per streaming multiprocessor.

To fully utilize both the block and thread slots,
to have 16 blocks, we should have

- $2,048/16 = 128$ threads per block, or
- $128/32 = 4$ warps per block.

interactions between resource limitations – P100

The P100 has 2,048 thread slots per streaming multiprocessor.
As $2,048 = 32 \times 64$, we have

$$\text{number of thread slots} = \text{warp size} \times \text{number of warps per block.}$$

For 32 threads per block, we have $2,048/32 = 64$ blocks
 \leftrightarrow at most 32 blocks per streaming multiprocessor.

To fully utilize both the block and thread slots,
to have 32 blocks, we should have

- $2,048/32 = 64$ threads per block, or
- $64/32 = 2$ warps per block.

streaming multiprocessor resources – part II

Comparing GPUs with respective compute capabilities 1.1, 2.0, 3.5, and 6.0: GeForce 9400M, Tesla C2050/C2070, K20C, and P100.

| compute capability | 1.1 | 2.0 | 3.5 | 6.0 |
|--|------|-------|------|------|
| number of 32-bit registers per streaming multiprocessor | 8K | 32K | 64K | |
| maximum amount of shared memory per streaming multiprocessor | 16KB | 48KB | 64KB | |
| number of shared memory banks | 16 | 32 | | |
| amount of local memory per thread | 16KB | 512KB | | |
| constant memory size | | 64KB | | |
| cache working set for constant memory per streaming memory | | 8KB | | 10KB |

Local memory resides in device memory, so local memory accesses have the same high latency and low bandwidth as global memory.

dynamic partitioning of resources

Registers hold frequently used programmer and compiler-generated variables to reduce access latency and conserve memory bandwidth.

Variables in a kernel that are not arrays are automatically placed into registers.

By dynamically partitioning the registers among blocks, a streaming multiprocessor can accommodate

- more blocks if they require few registers, and
- fewer blocks if they require many registers.

As with block and thread slots, there is a potential interaction between register limitations and other resource limitations.

interactions between resource limitations

Consider the matrix-matrix multiplication example. Assume

- the kernel uses 21 registers, and
- we have 16-by-16 thread blocks.

How many threads can run on each Streaming Multiprocessor (SM)?

- ① We calculate the number of registers for each block:

$$16 \times 16 \times 21 = 5,376 \text{ registers.}$$

- ② We have $32 \times 1,024$ registers per SM:

$$32 \times 1,024 / 5,376 = 6 \text{ blocks}$$

and $6 < 8$ = the maximum number of blocks per SM.

- ③ We calculate the number of threads per SM:

$$16 \times 16 \times 6 = 1,536 \text{ threads}$$

and we can have at most 1,536 threads per SM.

a performance cliff

Suppose we use one extra register, 22 instead of 21.

- ① We calculate the number of registers for each block:

$$16 \times 16 \times 22 = 5,632 \text{ registers.}$$

- ② We have $32 \times 1,024$ registers per SM:

$$32 \times 1,024 / 5,632 = 5 \text{ blocks.}$$

- ③ We calculate the number of threads per SM:

$$16 \times 16 \times 5 = 1,280 \text{ threads}$$

and with 21 registers we could use all 1,536 threads per SM.

Adding one register led to a reduction of 17% in the parallelism.

When a slight increase in one resource leads to a dramatic reduction in parallelism and performance, one speaks of a *performance cliff*.

Performance Considerations

1 Dynamic Partitioning of Resources

- streaming multiprocessor resources
- the CUDA occupancy calculator

2 the Compute Visual Profiler

- getting started with `computeprof`
- analysis of the kernel `matrixMul`

3 Data Prefetching and Instruction Mix

- registers between global and shared memory
- maximizing instruction throughput

spreadsheet in /usr/local/cuda/tools

CUDA GPU Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click): 3.5

1.b) Select Shared Memory Size Config (bytes): 49152

2.) Enter your resource usage:

Threads Per Block: 256
Registers Per Thread: 32
Shared Memory Per Block (bytes): 4096

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:

Active Threads per Multiprocessor: 2048

Active Warps per Multiprocessor: 64

Active Thread Blocks per Multiprocessor: 8

Occupancy of each Multiprocessor: 100%

Physical Limits for GPU Compute Capability:

3.5

Threads per Warp: 32

Warp per Multiprocessor: 64

Threads per Multiprocessor: 2048

Thread Blocks per Multiprocessor: 16

Total # of 32-bit registers per Multiprocessor: 65536

Register allocation unit size: 256

Register allocation granularity: warp

Registers per Thread: 256

Shared Memory per Multiprocessor (bytes): 49152

Shared Memory Allocation unit size: 256

Warp allocation granularity: 4

Maximum Thread Block Size: 1024

Allocated Resources = Allocatable

Warp (Threads Per Block / Threads Per Warp): 8 64 8

Registers (Warp limit per SM due to per-warp reg count): 8 64 8

Shared Memory (Bytes): 4096 49152 12

Note: SM is an abbreviation for (Single) Multiprocessor

Maximum Thread Blocks Per Multiprocessor: Blocks/SM * Warps/Block = Warps/SM

Limited by Max Warps or Max Blocks per Multiprocessor: 8 8 64

Limited by Registers per Multiprocessor: 8 8 64

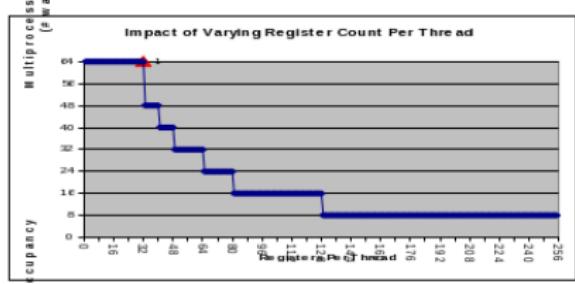
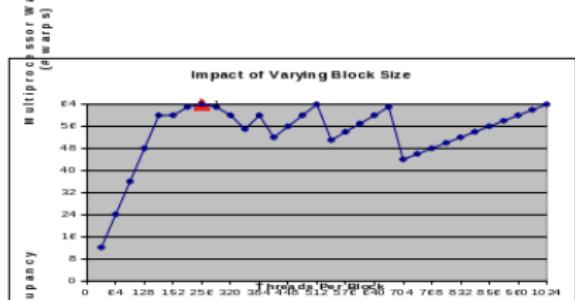
Limited by Shared Memory per Multiprocessor: 12 12 12

Note: Occupancy limit is shown in orange

Physical Max Warps/SM = 64
Occupancy = 64 / 64 = 100%

[Click Here for detailed instructions on how to use this occupancy calculator](#)
[For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda](http://developer.nvidia.com/cuda)

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.



Performance Considerations

1 Dynamic Partitioning of Resources

- streaming multiprocessor resources
- the CUDA occupancy calculator

2 the Compute Visual Profiler

- getting started with `computeprof`
- analysis of the kernel `matrixMul`

3 Data Prefetching and Instruction Mix

- registers between global and shared memory
- maximizing instruction throughput

getting started with computeprof

Compute Visual Profiler is a graphical user interface based profiling tool to measure performance and to find potential opportunities for optimization in order to achieve maximum performance.

Login to kepler with ssh -X and go the directory
/usr/local/cuda/bin/computeprof
to launch the program computeprof.

We look at one of the example projects matrixMul.

Performance Considerations

1 Dynamic Partitioning of Resources

- streaming multiprocessor resources
- the CUDA occupancy calculator

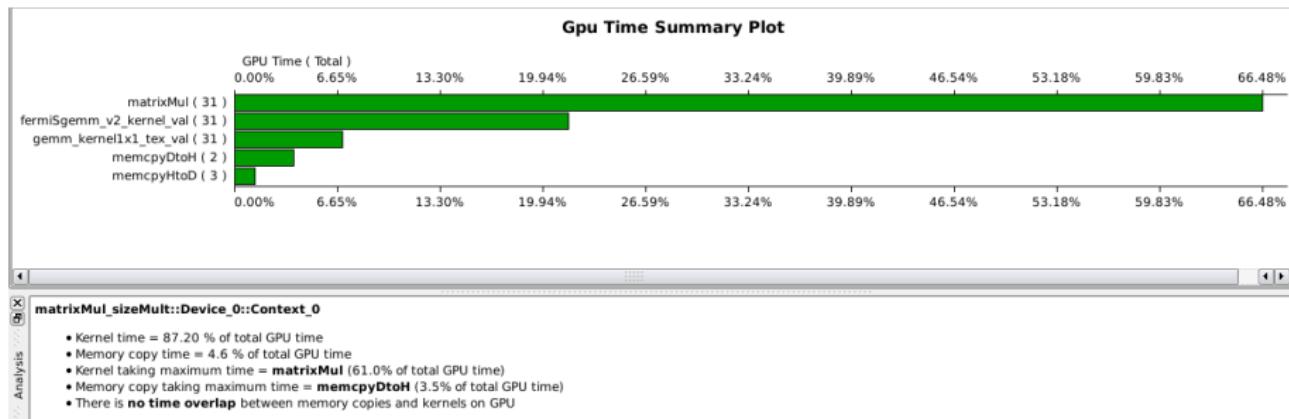
2 the Compute Visual Profiler

- getting started with `computeprof`
- analysis of the kernel `matrixMul`

3 Data Prefetching and Instruction Mix

- registers between global and shared memory
- maximizing instruction throughput

GPU time summary



limiting factor identification

Analysis for kernel matrixMul on device Tesla C2050

Summary profiling information for the kernel:

Number of calls: 31

Minimum GPU time(us): 4184.67

Maximum GPU time(us): 4192.67

Average GPU time(us): 4188.50

GPU time (%): 61.04

Grid size: [20 30 1]

Block size: [32 32 1]

Limiting Factor

Achieved Instruction Per Byte Ratio: 10.87 (Balanced Instruction Per Byte Ratio: 3.57)

Achieved Occupancy: 0.67 (Theoretical Occupancy: 0.67)

IPC: 1.02 (Maximum IPC: 2)

Achieved global memory throughput: 10.00 (Peak global memory throughput(GB/s): 144.00)

IPC = Instructions Per Cycle

memory throughput analysis

Memory Throughput Analysis for kernel matrixMul on device Tesla C2050

- Kernel requested global memory read throughput(GB/s): 23.47
- Kernel requested global memory write throughput(GB/s): 0.59
- Kernel requested global memory throughput(GB/s): 24.06
- L1 cache read throughput(GB/s): 23.47
- L1 cache global hit ratio (%): 0.00
- Texture cache memory throughput(GB/s): 0.00
- Texture cache hit rate(%): 0.00
- L2 cache texture memory read throughput(GB/s): 0.00
- L2 cache global memory read throughput(GB/s): 23.47
- L2 cache global memory write throughput(GB/s): 0.59
- L2 cache global memory throughput(GB/s): 24.06
- Local memory bus traffic(%): 0.00
- Global memory excess load(%): 0.00
- Global memory excess store(%): 0.00
- Achieved global memory read throughput(GB/s): 9.27
- Achieved global memory write throughput(GB/s): 0.73
- Achieved global memory throughput(GB/s): 10.00
- Peak global memory throughput(GB/s): 144.00

instruction throughput analysis

Instruction Throughput Analysis for kernel matrixMul on device Tesla C2050

- IPC: 1.02
- Maximum IPC: 2
- Divergent branches(%): 0.00
- Control flow divergence(%): 0.04
- Replayed Instructions(%): 0.57
 - Global memory replay(%): 2.25
 - Local memory replays(%): 0.00
 - Shared bank conflict replay(%): 0.00
- Shared memory bank conflict per shared memory instruction(%): 0.00

IPC = Instructions Per Cycle

occupancy analysis

Occupancy Analysis for kernel matrixMul on device Tesla C2050

- Kernel details: Grid size: [20 30 1], Block size: [32 32 1]
- Register Ratio: 0.8125 (26624 / 32768) [25 registers per thread]
- Shared Memory Ratio: 0.166667 (8192 / 49152) [8192 bytes per Block]
- Active Blocks per SM: 1 (Maximum Active Blocks per SM: 8)
- Active threads per SM: 1024 (Maximum Active threads per SM: 1536)
- Potential Occupancy: 0.666667 (32 / 48)
- Occupancy limiting factor: Block-Size

Performance Considerations

1 Dynamic Partitioning of Resources

- streaming multiprocessor resources
- the CUDA occupancy calculator

2 the Compute Visual Profiler

- getting started with `computeprof`
- analysis of the kernel `matrixMul`

3 Data Prefetching and Instruction Mix

- registers between global and shared memory
- maximizing instruction throughput

accessing global memory

One of the most important resource limitations is access to global memory and long latencies.

Scheduling other warps while waiting for memory access is powerful, but often not enough.

A complementary to warp scheduling solution is to prefetch the next data elements while processing the current data elements.

Combined with tiling, data prefetching provides extra independent instructions to enable the scheduling of more warps to tolerate long memory access latencies.

prefetching in registers

For the tiled matrix-matrix multiplication,
the code below combines prefetching with tiling:

```
load first tile from global memory into registers;  
loop  
{  
    deposit tile from registers to shared memory;  
    __syncthreads();  
    load next tile from global memory into registers;  
    process current tile;  
    __syncthreads();  
}
```

The prefetching adds independent instructions between loading the data from global memory and processing the data.

Performance Considerations

1 Dynamic Partitioning of Resources

- streaming multiprocessor resources
- the CUDA occupancy calculator

2 the Compute Visual Profiler

- getting started with `computeprof`
- analysis of the kernel `matrixMul`

3 Data Prefetching and Instruction Mix

- registers between global and shared memory
- maximizing instruction throughput

throughput of arithmetic instructions

Number of operations per clock cycle per multiprocessor:

| compute capability | 1.x | 2.0 | 3.5 | 6.0 |
|---|-----|-----|-----|-----|
| 32-bit floating-point add, multiply, multiply-add | 8 | 32 | 192 | 64 |
| 64-bit floating-point add, multiply, multiply-add | 1 | 16 | 64 | 4 |
| 32-bit integer add, logical operation, shift, compare | 8 | 32 | 160 | 128 |
| 32-bit floating-point reciprocal, square root, log, exp, sine, cosine | 2 | 4 | 32 | 32 |

loop unrolling

Consider the following code snippet:

```
for(int k = 0; k < m; k++)
    C[i][j] += A[i][k]*B[k][j];
```

Counting all instructions:

- 1 loop branch instruction ($k < m$);
- 1 loop counter update instruction ($k++$);
- 3 address arithmetic instructions ($[i][j]$, $[i][k]$, $[k][j]$);
- 2 floating-point arithmetic instructions (+ and *).

Of the 7 instructions, only 2 are floating point.

Loop unrolling reduces the number of loop branch instructions, loop counter updates, address arithmetic instructions.

Note: `gcc -funroll-loops` is enabled with `gcc -O2`.

summary and exercises

We covered §6.3, §6.4, and §6.5 in the book of Kirk & Hwu;
using data from Appendix G in the CUDA programming Guide.

- 1 Examine the occupancy calculator for the graphics card on your laptop or desktop.
- 2 Read the user guide of the compute visual profiler and perform a run on GPU code you wrote (of some previous exercise or your code for the third project). Explain the analysis of the kernel.
- 3 Redo the first “interactions between resource limitations” of this lecture using the specifications for compute capability 1.1.
- 4 Redo the second “interactions between resource limitations” of this lecture using the specifications for compute capability 1.1.

Quad Doubles on a GPU

1 Floating-Point Arithmetic

- floating-point numbers
- quad double arithmetic
- quad doubles for use in CUDA programs

2 Quad Double Square Roots

- quad double arithmetic on a GPU
- a kernel using `gqd_real`
- performance considerations

MCS 572 Lecture 37
Introduction to Supercomputing
Jan Verschelde, 16 November 2016

Quad Doubles on a GPU

1 Floating-Point Arithmetic

- floating-point numbers
- quad double arithmetic
- quad doubles for use in CUDA programs

2 Quad Double Square Roots

- quad double arithmetic on a GPU
- a kernel using `gqd_real`
- performance considerations

floating-point numbers

A floating-point number consists of a sign bit, exponent, and a fraction (also known as the mantissa).

Almost all microprocessors follow the IEEE 754 standard.

GPU hardware supports 32-bit (single float)
and for compute capability ≥ 1.3 also double floats.

Numerical analysis studies algorithms for continuous problems,
investigating

- problems for their sensitivity to errors in the input; and
- algorithms for their propagation of roundoff errors.

parallel numerical algorithms

The floating-point addition is *not* associative!

Parallel algorithms compute and accumulate the results in an order that is different from their sequential versions.

Example: Adding a sequence of numbers is more accurate if the numbers are sorted in increasing order.

Instead of speedup, we can ask questions about quality up:

- If we can afford to keep the total running time constant, does a faster computer give us more accurate results?
- How many more processors do we need to guarantee a result?

Quad Doubles on a GPU

1 Floating-Point Arithmetic

- floating-point numbers
- **quad double arithmetic**
- quad doubles for use in CUDA programs

2 Quad Double Square Roots

- quad double arithmetic on a GPU
- a kernel using `gqd_real`
- performance considerations

quadruple precision

A quad double is an unevaluated sum of 4 doubles,
improves working precision from 2.2×10^{-16} to 2.4×10^{-63} .

Y. Hida, X.S. Li, and D.H. Bailey: **Algorithms for quad-double precision floating point arithmetic.** In *15th IEEE Symposium on Computer Arithmetic* pages 155–162. IEEE, 2001. Software at
<http://crd.lbl.gov/~dhbailey/mpdist>.

A quad double builds on double double, some features:

- The least significant part of a double double can be interpreted as a compensation for the roundoff error.
- Predictable overhead: working with double double is of the same cost as working with complex numbers.

Newton's method for \sqrt{x}

```
#include <iostream>
#include <iomanip>
#include <qd/qd_real.h>
using namespace std;

qd_real newton ( qd_real x )
{
    qd_real y = x;
    for(int i=0; i<10; i++)
        y -= (y*y - x) / (2.0*y);
    return y;
}
```

the main program

```
int main ( int argc, char *argv[] )
{
    cout << "give x : ";
    qd_real x; cin >> x;
    cout << setprecision(64);
    cout << "           x : " << x << endl;

    qd_real y = newton(x);
    cout << "   sqrt(x) : " << y << endl;

    qd_real z = y*y;
    cout << "sqrt(x)^2 : " << z << endl;

    return 0;
}
```

the makefile

If the program is on file `newton4sqrt.cpp`
and the makefile contains

```
QD_ROOT=/usr/local/qd-2.3.17  
QD_LIB=/usr/local/lib
```

`newton4sqrt:`

```
    g++ -I$(QD_ROOT)/include newton4sqrt.cpp \  
        $(QD_LIB)/libqd.a -o /tmp/newton4sqrt
```

then we can create the executable as

```
$ make newton4sqrt  
g++ -I/usr/local/qd-2.3.17/include newton4sqrt.cpp \  
    /usr/local/lib/libqd.a -o /tmp/newton4sqrt  
$
```

Quad Doubles on a GPU

1 Floating-Point Arithmetic

- floating-point numbers
- quad double arithmetic
- **quad doubles for use in CUDA programs**

2 Quad Double Square Roots

- quad double arithmetic on a GPU
- a kernel using `gqd_real`
- performance considerations

extended precision on the GPU

Large problems often need extra precision.

The QD library has been ported to the GPU.

Mian Lu, Bingsheng He, and Qiong Luo: **Supporting extended precision on graphics processors**. In A. Ailamaki and P.A. Boncz, editors, *Proceedings of the Sixth International Workshop on Data Management on New Hardware (DaMoN 2010), June 7, 2010, Indianapolis, Indiana*, pages 19–26, 2010.

Software at <http://code.google.com/p/gpuprec/>, and at <https://github.com/lumianph/gpuprec/tree/master/gqd>.

Installed on `kepler` and `pascal` in `/usr/local/gqd_1_2`.

For graphics cards of compute capability < 1.3, one could use the freely available Cg software of Andrew Thall to achieve double precision using float-float arithmetic.

gqd_reals are of double4 type

```
#include "gqd_type.h"
#include "vector_types.h"
#include <qd/qd_real.h>

void qd2gqd ( qd_real *a, gqd_real *b )
{
    b->x = a->x[0];
    b->y = a->x[1];
    b->z = a->x[2];
    b->w = a->x[3];
}

void gqd2qd ( gqd_real *a, qd_real *b )
{
    b->x[0] = a->x;
    b->x[1] = a->y;
    b->x[2] = a->z;
    b->x[3] = a->w;
}
```

a first kernel

```
#include "gqd.cu"

__global__ void testdiv2 ( gqd_real *x, gqd_real *y )
{
    *y = *x/2.0;
}

int divide_by_two ( gqd_real *x, gqd_real *y )
{
    gqd_real *xdevice;
    size_t s = sizeof(gqd_real);
    cudaMalloc((void**)&xdevice,s);
    cudaMemcpy(xdevice,x,s,cudaMemcpyHostToDevice);
    gqd_real *ydevice;
    cudaMalloc((void**)&ydevice,s);
    testdiv2<<<1,1>>>(xdevice,ydevice);
    cudaMemcpy(y,ydevice,s,cudaMemcpyDeviceToHost);
    return 0;
}
```

testing the first kernel

```
#include <iostream>
#include <iomanip>
#include "gqd_type.h"
#include "first_gqd_kernel.h"
#include "gqd_qd_util.h"
#include <qd/qd_real.h>
using namespace std;

int main ( int argc, char *argv[] )
{
    qd_real qd_x = qd_real::pi;
    gqd_real x;
    qd2gqd(&qd_x, &x);
    gqd_real y;

    cout << " x : " << setprecision(64) << qd_x << endl;
```

test program continued

```
int fail = divide_by_two(&x, &y);  
  
qd_real qd_y;  
gqd2qd(&y, &qd_y);  
  
if(fail == 0) cout << " y : " << qd_y << endl;  
  
cout << "2y : " << 2.0*qd_y << endl;  
  
return 0;  
}
```

the makefile

```
QD_ROOT=/usr/local/qd-2.3.17
QD_LIB=/usr/local/lib
GQD_HOME=/usr/local/gqd_1_2
SDK_HOME=/usr/local/cuda/sdk

test_pi2_gqd_kernel:
    @-echo ">>> compiling kernel ..."
    nvcc -I$(GQD_HOME)/inc -I$(SDK_HOME)/C/common/inc \
        -c first_gqd_kernel.cu
    @-echo ">>> compiling utilities ..."
    g++ -I/usr/local/cuda/include -I$(GQD_HOME)/inc \
        -I$(QD_ROOT)/include -c gqd_qd_util.cpp
    @-echo ">>> compiling test program ..."
    g++ test_pi2_gqd_kernel.cpp -c \
        -I/usr/local/cuda/include -I$(GQD_HOME)/inc \
        -I$(QD_ROOT)/include
    @-echo ">>> linking ..."
    g++ -I$(GQD_HOME)/inc -I$(QD_ROOT)/include \
        first_gqd_kernel.o test_pi2_gqd_kernel.o gqd_qd_util.o \
        $(QD_LIB)/libqgd.a \
        -o /tmp/test_pi2_gqd_kernel \
        -lcuda -lcutil_x86_64 -lcudart \
        -L/usr/local/cuda/lib64 -L$(SDK_HOME)/C/lib
```

compiling and running

```
$ make test_pi2_gqd_kernel
>>> compiling kernel ...
nvcc -I/usr/local/gqd_1_2/inc -I/usr/local/cuda/sdk/C/common/inc \
      -c first_gqd_kernel.cu
>>> compiling utilities ...
g++ -I/usr/local/cuda/include -I/usr/local/gqd_1_2/inc \
      -I/usr/local/qd-2.3.17/include -c gqd_qd_util.cpp
>>> compiling test program ...
g++ test_pi2_gqd_kernel.cpp -c \
      -I/usr/local/cuda/include -I/usr/local/gqd_1_2/inc \
      -I/usr/local/qd-2.3.17/include
>>> linking ...
g++ -I/usr/local/gqd_1_2/inc -I/usr/local/qd-2.3.17/include \
      first_gqd_kernel.o test_pi2_gqd_kernel.o gqd_qd_util.o \
      /usr/local/lib/libqd.a \
      -o /tmp/test_pi2_gqd_kernel \
      -lcuda -lcutil_x86_64 -lcudart \
      -L/usr/local/cuda/lib64 -L/usr/local/cuda/sdk/C/lib
$ /tmp/test_pi2_gqd_kernel
x : 3.1415926535897932384626433832795028841971693993751058209749445923e+00
y : 1.5707963267948966192313216916397514420985846996875529104874722961e+00
2y : 3.1415926535897932384626433832795028841971693993751058209749445923e+00
$
```

Quad Doubles on a GPU

1 Floating-Point Arithmetic

- floating-point numbers
- quad double arithmetic
- quad doubles for use in CUDA programs

2 Quad Double Square Roots

- quad double arithmetic on a GPU
- a kernel using `gqd_real`
- performance considerations

quad double arithmetic on a GPU

Recall our first CUDA program to take the square root of complex numbers stored in a `double2` array.

In using quad doubles on a GPU, we have 3 stages:

- ① The kernel in a file with extension `.cu` is compiled with `nvcc -c` into an object file.
- ② The application code is compiled with `g++ -c`.
- ③ The linker `g++` takes `.o` files and libraries on input to make an executable file.

Working without a makefile now becomes very tedious.

the makefile

```
QD_ROOT=/usr/local/qd-2.3.17
QD_LIB=/usr/local/lib
GQD_HOME=/usr/local/gqd_1_2
SDK_HOME=/usr/local/cuda/sdk

sqrt_gqd_kernel:
    @-echo ">>> compiling kernel ..."
    nvcc -I$(GQD_HOME)/inc -I$(SDK_HOME)/C/common/inc \
        -c sqrt_gqd_kernel.cu
    @-echo ">>> compiling utilities ..."
    g++ -I/usr/local/cuda/include -I$(GQD_HOME)/inc \
        -I$(QD_ROOT)/include -c gqd_qd_util.cpp
    @-echo ">>> compiling test program ..."
    g++ run_sqrt_gqd_kernel.cpp -c \
        -I/usr/local/cuda/include -I$(GQD_HOME)/inc \
        -I$(QD_ROOT)/include
    @-echo ">>> linking ..."
    g++ -I$(GQD_HOME)/inc -I$(QD_ROOT)/include \
        sqrt_gqd_kernel.o run_sqrt_gqd_kernel.o gqd_qd_util.o \
        $(QD_LIB)/libqfd.a \
        -o /tmp/run_sqrt_gqd_kernel \
        -lcuda -lcutil_x86_64 -lcudart \
        -L/usr/local/cuda/lib64 -L$(SDK_HOME)/C/lib
```

running make

```
$ make sqrt_gqd_kernel
>>> compiling kernel ...
nvcc -I/usr/local/gqd_1_2/inc -I/usr/local/cuda/sdk/C/common/inc \
      -c sqrt_gqd_kernel.cu
>>> compiling utilities ...
g++ -I/usr/local/cuda/include -I/usr/local/gqd_1_2/inc \
      -I/usr/local/qd-2.3.17/include -c gqd_qd_util.cpp
>>> compiling test program ...
g++ run_sqrt_gqd_kernel.cpp -c \
      -I/usr/local/cuda/include -I/usr/local/gqd_1_2/inc \
      -I/usr/local/qd-2.3.17/include
>>> linking ...
g++ -I/usr/local/gqd_1_2/inc -I/usr/local/qd-2.3.17/include \
      sqrt_gqd_kernel.o run_sqrt_gqd_kernel.o gqd_qd_util.o \
      /usr/local/lib/libqfd.a \
      -o /tmp/run_sqrt_gqd_kernel \
      -lcuda -lcututil_x86_64 -lcudart \
      -L/usr/local/cuda/lib64 -L/usr/local/cuda/sdk/C/lib
$
```

Quad Doubles on a GPU

1 Floating-Point Arithmetic

- floating-point numbers
- quad double arithmetic
- quad doubles for use in CUDA programs

2 Quad Double Square Roots

- quad double arithmetic on a GPU
- **a kernel using `gqd_real`**
- performance considerations

a kernel using gqd_real

```
#include "gqd.cu"

__global__ void sqrtNewton ( gqd_real *x, gqd_real *y )
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    gqd_real c = x[i];
    gqd_real r = c;
    for(int j=0; j<10; j++)
        r = r - (r*r - c)/(2.0*r);
    y[i] = r;
}
```

the file sqrt_gqd_kernel.cu continued

```
int sqrt_by_Newton ( int n, gqd_real *x, gqd_real *y )
{
    gqd_real *xdevice;
    size_t s = n*sizeof(gqd_real);
    cudaMalloc((void**)&xdevice,s);
    cudaMemcpy(xdevice,x,s,cudaMemcpyHostToDevice);

    gqd_real *ydevice;
    cudaMalloc((void**)&ydevice,s);

    sqrtNewton<<<n/32,32>>>(xdevice,ydevice);

    cudaMemcpy(y,ydevice,s,cudaMemcpyDeviceToHost);

    return 0;
}
```

the main program

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include "gqd_type.h"
#include "sqrt_gqd_kernel.h"
#include "gqd_qd_util.h"
#include <qd/qd_real.h>
using namespace std;

int main ( int argc, char *argv[] )
{
    const int n = 256;
    gqd_real *x = (gqd_real*)calloc(n,sizeof(gqd_real));
    gqd_real *y = (gqd_real*)calloc(n,sizeof(gqd_real));
```

run_sqrt_gqd_kernel.cpp continued

```
for(int i = 0; i<n; i++)
{
    x[i].x = (double) (i+2);
    x[i].y = 0.0; x[i].z = 0.0; x[i].w = 0.0;
}
int fail = sqrt_by_Newton(n,x,y);
if(fail == 0)
{
    const int k = 24;
    qd_real qd_x;
    gqd2qd(&x[k],&qd_x);
    qd_real qd_y;
    gqd2qd(&y[k],&qd_y);
    cout << "      x      : " << setprecision(64) << qd_x << endl;
    cout << "sqrt(x)      : " << setprecision(64) << qd_y << endl;
    cout << "sqrt(x)^2   : " << setprecision(64) << qd_y*qd_y
        << endl;
}
return 0;
}
```

Quad Doubles on a GPU

1 Floating-Point Arithmetic

- floating-point numbers
- quad double arithmetic
- quad doubles for use in CUDA programs

2 Quad Double Square Roots

- quad double arithmetic on a GPU
- a kernel using `gqd_real`
- performance considerations

sequential and interval memory layout

Consider four quad doubles a , b , c , and d .

Stored in a sequential memory layout:

| | | | | | | | | | | | | | | | |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| a_0 | a_1 | a_2 | a_3 | b_0 | b_1 | b_2 | b_3 | c_0 | c_1 | c_2 | c_3 | d_0 | d_1 | d_2 | d_3 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

Stored in an interval memory layout:

| | | | | | | | | | | | | | | | |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| a_0 | b_0 | c_0 | d_0 | a_1 | b_1 | c_1 | d_1 | a_2 | b_2 | c_2 | d_2 | a_3 | b_3 | c_3 | d_3 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

The implementation with an interval memory layout is reported to be three times faster over the sequential memory layout.

Bibliography

- A. Thall. **Extended-Precision Floating-Point Numbers for GPU Computation.** Software available at andrewthall.org.
- Y. Hida, X.S. Li, and D.H. Bailey. **Algorithms for quad-double precision floating point arithmetic.** In *15th IEEE Symposium on Computer Arithmetic*, pages 155–162. IEEE, 2001.
Software at <http://crd.lbl.gov/~dhbailey/mpdist>.
- M. Lu, B. He, and Q. Luo. **Supporting extended precision on graphics processors.** In A. Ailamaki and P.A. Boncz, editors, *Proceedings of the Sixth International Workshop on Data Management on New Hardware (DaMoN 2010), June 7, 2010, Indianapolis, Indiana*, pages 19–26, 2010.
Software at <http://code.google.com/p/gpuprec/> and at <https://github.com/lumianph/gpuprec/tree/master/gqd>.

summary and exercises

Chapter 7 in the book of Kirk & Hwu provides some background.
The application of quad double arithmetic is an illustration of combined usage of `nvcc` and `g++` to compile and link several libraries.

Some exercises:

- 1 Compare the performance of the CUDA program for Newton's method for square root with quad doubles to the code of lecture 29.
- 2 Extend the code so it works for complex quad double arithmetic.
- 3 Use quad doubles to implement the second parallel sum algorithm of lecture 33. Could the parallel implementation with quad doubles run as fast as sequential code with doubles?
- 4 Consider the program to approximate π of lecture 13. Write a version for the GPU and compare the performance with the multicore version of lecture 13.

Concurrent Kernels and Multiple GPUs

1 Page Locked Host Memory

- host memory that is page locked or pinned
- executing a zero copy

2 Concurrent Kernels

- streams and concurrency
- squaring numbers with concurrent kernels

3 Using Multiple GPUs

- counting the number of available devices
- working with multiple GPUs

MCS 572 Lecture 39
Introduction to Supercomputing
Jan Verschelde, 21 November 2016

Concurrent Kernels and Multiple GPUs

1 Page Locked Host Memory

- host memory that is page locked or pinned
- executing a zero copy

2 Concurrent Kernels

- streams and concurrency
- squaring numbers with concurrent kernels

3 Using Multiple GPUs

- counting the number of available devices
- working with multiple GPUs

page locked or pinned memory

In contrast to regular pageable host memory, the runtime provides functions to allocate (and free) *page locked* memory.

Another name for memory that is page locked is *pinned*.

Using page locked memory has several benefits:

- Copies between page locked memory and device memory can be performed concurrently with kernel execution.
- Page locked host memory can be mapped into the address space of the device, eliminating the need to copy, we say *zero copy*.
- Bandwidth between page locked host memory and device may be higher.

Page locked host memory is a scarce resource.

The NVIDIA CUDA Best Practices Guide assigns a low priority to zero-copy operations (i.e.: mapping host memory to the device).

allocating and mapping pinned host memory

To allocate page locked memory, we use `cudaHostAlloc()` and to free the memory, we call `cudaFreeHost()`.

To map host memory on the device:

- The flag `cudaHostAllocMapped` must be given to `cudaHostAlloc()` when allocating host memory.
- A call to `cudaHostGetDevicePointer()` maps the host memory to the device.

If all goes well, then no copies from host to device memory and from device to host memory are needed.

Not all devices support pinned memory, it is recommended practice to check the device properties (see the `deviceQuery` in the SDK).

Concurrent Kernels and Multiple GPUs

1 Page Locked Host Memory

- host memory that is page locked or pinned
- executing a zero copy

2 Concurrent Kernels

- streams and concurrency
- squaring numbers with concurrent kernels

3 Using Multiple GPUs

- counting the number of available devices
- working with multiple GPUs

using pinned memory

```
$ /tmp/pinnedmemoryuse  
Tesla K20c supports mapping host memory.
```

The error code of cudaHostAlloc : 0

```
Squaring 32 numbers 1 2 3 4 5 6 7 8 9 10 11 12 13 \  
14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32...
```

The fail code of cudaHostGetDevicePointer : 0

```
After squaring 32 numbers 1 4 9 16 25 36 49 64 81 \  
100 121 144 169 196 225 256 289 324 361 400 441 484 \  
529 576 625 676 729 784 841 900 961 1024...
```

\$

the program pinnedmemoryuse.cu

```
/* This program illustrates pinned (page locked) memory. */

#include <stdio.h>

int checkDeviceProp ( cudaDeviceProp p );
/*
 * Returns 0 if the device does not support mapping
 * host memory, returns 1 otherwise. */

```

checking the device

```
int checkDeviceProp ( cudaDeviceProp p )
{
    int support = p.canMapHostMemory;

    if(support == 0)
        printf("%s does not support mapping host memory.\n",
               p.name);
    else
        printf("%s supports mapping host memory.\n", p.name);

    return support;
}
```

a simple kernel

```
__global__ void Square ( float *x )
/*
 * A kernel where the i-th thread squares x[i]
 * and stores the result in x[i]. */
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    x[i] = x[i]*x[i];
}
```

the main program

```
void square_with_pinned_memory ( int n );
/*
 * Illustrates the use of pinned memory to square
 * a sequence of n numbers. */

int main ( int argc, char* argv[] )
{
    cudaDeviceProp dev;

    cudaGetDeviceProperties(&dev, 0);

    int success = checkDeviceProp(dev);

    if(success != 0)
        square_with_pinned_memory(32);

    return 0;
}
```

allocating pinned memory

```
void square_with_pinned_memory ( int n )
{
    float *xhost;
    size_t sz = n*sizeof(float);
    int error = cudaHostAlloc((void**)&xhost,
                              sz,cudaHostAllocMapped);
    printf("\nThe error code of cudeHostAlloc : %d\n",
           error);

    for(int i=0; i<n; i++) xhost[i] = (float) (i+1);
    printf("\nSquaring %d numbers",n);
    for(int i=0; i<n; i++) printf(" %d", (int) xhost[i]);
    printf("...\n\n");
```

mapping host memory

```
float *xdevice;

int fail = cudaHostGetDevicePointer
            ((void**)&xdevice, (void*)xhost, 0);
printf("\nThe fail code of cudaHostGetDevicePointer : \
      %d\n", fail);

Square<<<1, n>>>(xdevice);

cudaDeviceSynchronize();

printf("\nAfter squaring %d numbers", n);
for(int i=0; i<n; i++) printf(" %d", (int) xhost[i]);
printf("...\n\n");

cudaFreeHost(xhost);
}
```

Concurrent Kernels and Multiple GPUs

1 Page Locked Host Memory

- host memory that is page locked or pinned
- executing a zero copy

2 Concurrent Kernels

- streams and concurrency
- squaring numbers with concurrent kernels

3 Using Multiple GPUs

- counting the number of available devices
- working with multiple GPUs

streams and concurrency

The Fermi architecture supports the simultaneous execution of kernels.

Benefits:

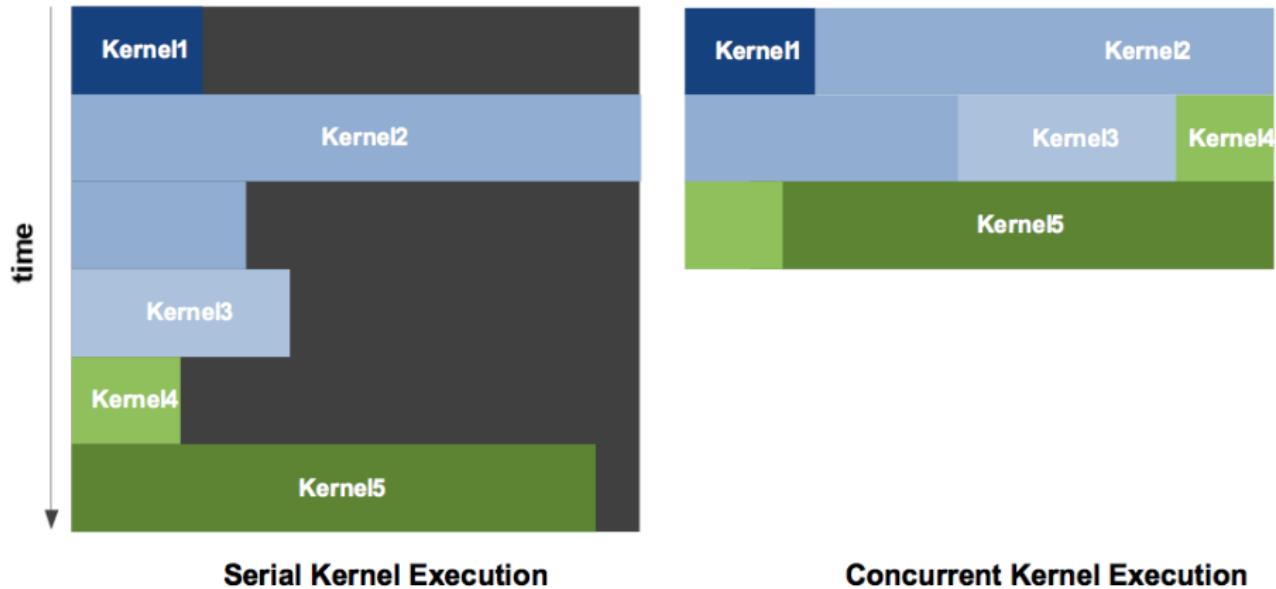
- Simultaneous execution of small kernels utilize whole GPU.
- Overlapping kernel execution with device to host memory copy.

A *stream* is a sequence of commands that execute in order.

Different streams may execute concurrently.

The maximum number of kernel launches that a device can execute concurrently is four.

concurrent kernel execution



from the NVIDIA Fermi Compute Architecture Whitepaper

concurrent copy and kernel execution (4 streams)

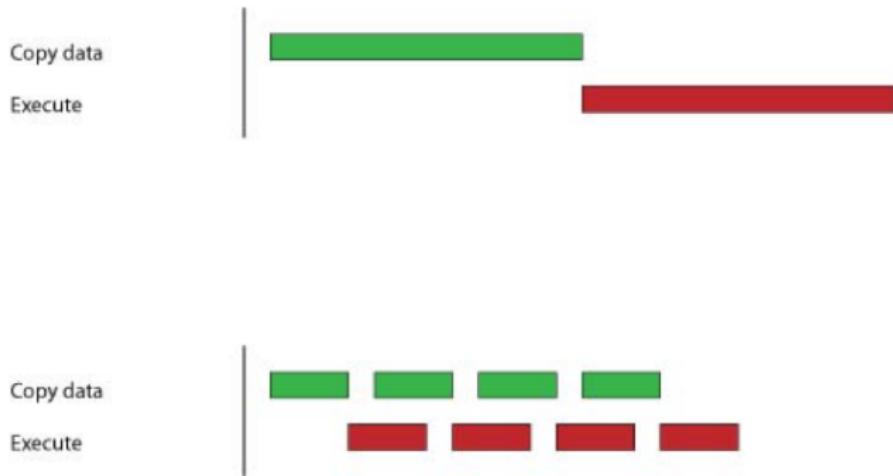


Figure 3.1 Timeline Comparison for Sequential (top) and Concurrent (bottom) Copy and Kernel Execution

from the NVIDIA CUDA Best Practices Guide

Concurrent Kernels and Multiple GPUs

1 Page Locked Host Memory

- host memory that is page locked or pinned
- executing a zero copy

2 Concurrent Kernels

- streams and concurrency
- squaring numbers with concurrent kernels

3 Using Multiple GPUs

- counting the number of available devices
- working with multiple GPUs

squaring numbers

```
$ /tmp/concurrent
Tesla K20c supports concurrent kernels
compute capability : 3.5
number of multiprocessors : 13

Launching 4 kernels on 16 numbers 1 2 3 4 5 6 7 8 9 10 \
11 12 13 14 15 16...
the 16 squared numbers are 1 4 9 16 25 36 49 64 81 100 \
121 144 169 196 225 256
$
```

a simple kernel

```
__global__ void Square ( float *x, float *y )
/*
 * A kernel where the i-th thread squares x[i]
 * and stores the result in y[i]. */
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    y[i] = x[i]*x[i];
}
```

checking if concurrency is supported

```
int checkDeviceProp ( cudaDeviceProp p )
{
    int support = p.concurrentKernels;

    if(support == 0)
        printf("%s does not support concurrent kernels\n",
               p.name);
    else
        printf("%s supports concurrent kernels\n",p.name);

    printf("  compute capability : %d.%d \n",
           p.major,p.minor);
    printf("  number of multiprocessors : %d \n",
           p.multiProcessorCount);

    return support;
}
```

the main program

```
void launchKernels ( void );
/*
 * Launches concurrent kernels on arrays of floats. */

int main ( int argc, char* argv[] )
{
    cudaDeviceProp dev;
    cudaGetDeviceProperties(&dev, 0);

    int success = checkDeviceProp(dev);
    if(success != 0) launchKernels();

    return 0;
}
```

allocating memories

```
void launchKernels ( void )
{
    const int nbstreams = 4;
    const int chunk = 4;
    const int nbdata = chunk*nbstreams;

    float *xhost;
    size_t sz = nbdata*sizeof(float);

    cudaMallocHost((void**)&xhost,sz);

    for(int i=0; i<nbdata; i++) xhost[i] = (float) (i+1);
    printf("\nLaunching %d kernels on %d numbers",
          nbstreams,nbdata);
    for(int i=0; i<nbdata; i++)
        printf(" %d", (int) xhost[i]);
    printf("...\n\n");
    float *xdevice; cudaMalloc((void**)&xdevice,sz);
    float *ydevice; cudaMalloc((void**)&ydevice,sz);
```

asynchronous concurrent execution

```
cudaStream_t s[nbstreams];
for(int i=0; i<nbstreams; i++) cudaStreamCreate(&s[i]);

for(int i=0; i<nbstreams; i++)
    cudaMemcpyAsync
        (&xdevice[i*chunk], &xhost[i*chunk],
         sz/nbstreams, cudaMemcpyHostToDevice, s[i]);

for(int i=0; i<nbstreams; i++)
    Square<<<1,chunk,0,s[i]>>>
        (&xdevice[i*chunk], &ydevice[i*chunk]);
```

synchronization

```
for(int i=0; i<nbstreams; i++)
    cudaMemcpyAsync
        (&xhost[i*chunk], &ydevice[i*chunk],
         sz/nbstreams, cudaMemcpyDeviceToHost, s[i]);

cudaDeviceSynchronize();

printf("the %d squared numbers are", nbdata);
for(int i=0; i<nbdata; i++)
    printf(" %d", (int) xhost[i]);
printf("\n");

for(int i=0; i<nbstreams; i++) cudaStreamDestroy(s[i]);
cudaFreeHost(xhost);
cudaFree(xdevice); cudaFree(ydevice);
}
```

Concurrent Kernels and Multiple GPUs

1 Page Locked Host Memory

- host memory that is page locked or pinned
- executing a zero copy

2 Concurrent Kernels

- streams and concurrency
- squaring numbers with concurrent kernels

3 Using Multiple GPUs

- counting the number of available devices
- working with multiple GPUs

enumerating available devices

```
$ /tmp/count_devices
number of devices : 3
graphics card 0 :
    name : Tesla K20c
    number of multiprocessors : 13
graphics card 1 :
    name : GeForce GT 620
    number of multiprocessors : 2
graphics card 2 :
    name : Tesla K20c
    number of multiprocessors : 13
$
```

the program count_devices.cu

```
/* This program illustrates counting available devices,
 * compile it with nvcc and note the extension .cu
 * and for a more detailed version, see deviceQuery.cpp
 * of the GPU Computing SDK */

#include <stdio.h>

void printDeviceProp ( cudaDeviceProp p )
/*
 * prints some device properties */
{
    printf(" name : %s \n",p.name);
    printf(" number of multiprocessors : %d \n",
           p.multiProcessorCount);
}
```

the main program

```
int main ( int argc, char* argv[] )
{
    int deviceCount;
    cudaGetDeviceCount (&deviceCount);
    printf("number of devices : %d\n",deviceCount);

    for(int d = 0; d < deviceCount; d++)
    {
        cudaDeviceProp dev;
        cudaGetDeviceProperties (&dev,d);
        printf("graphics card %d :\n",d);
        printDeviceProp (dev);
    }

    return 0;
}
```

Concurrent Kernels and Multiple GPUs

1 Page Locked Host Memory

- host memory that is page locked or pinned
- executing a zero copy

2 Concurrent Kernels

- streams and concurrency
- squaring numbers with concurrent kernels

3 Using Multiple GPUs

- counting the number of available devices
- working with multiple GPUs

computing with multiple GPUs

Chapter 8 of the NVIDIA CUDA Best Practices Guide describes multi-GPU programming.

To work with p GPUs concurrently, the CPU can use

- p lightweight threads (Pthreads, OpenMP, etc); or
- p heavyweight threads (or processes) with MPI.

The command to select a GPU is `cudaSetDevice()`.

All inter-GPU communication happens through the host.

See the `simpleMultiGPU` of the GPU Computing SDK.

Case Study: Advanced MRI Reconstruction

1 an Application Case Study

- magnetic resonance imaging
- iterative reconstruction

2 Acceleration on GPU

- determining the kernel parallelism structure
- loop splitting
- loop interchange
- using registers to reduce memory accesses
- chunking data to fit into constant memory
- using hardware trigonometry functions

MCS 572 Lecture 38
Introduction to Supercomputing
Jan Verschelde, 18 November 2016

Advanced MRI Reconstruction

1 an Application Case Study

- magnetic resonance imaging
- iterative reconstruction

2 Acceleration on GPU

- determining the kernel parallelism structure
- loop splitting
- loop interchange
- using registers to reduce memory accesses
- chunking data to fit into constant memory
- using hardware trigonometry functions

magnetic resonance imaging

Magnetic Resonance Imaging (MRI) is a safe and noninvasive probe of the structure and function of tissues in the body.

MRI consists of two phases:

- ① Acquisition or scan: the scanner samples data in the spatial-frequency domain along a predefined trajectory.
- ② Reconstruction of the samples into an image.

Limitations: noise, imaging artifacts, long acquisition times.

Three often conflicting goals:

- Short scan time to reduce patient discomfort.
- High resolution and fidelity for early detection.
- High signal-to-noise ratio (SNR).

Massively parallel computing provides disruptive breakthrough.

problem formulation

The reconstructed image $m(\mathbf{r})$ is

$$\hat{m}(\mathbf{r}) = \sum_j W(\mathbf{k}_j) s(\mathbf{k}_j) e^{i2\pi\mathbf{k}_j \cdot \mathbf{r}}$$

where

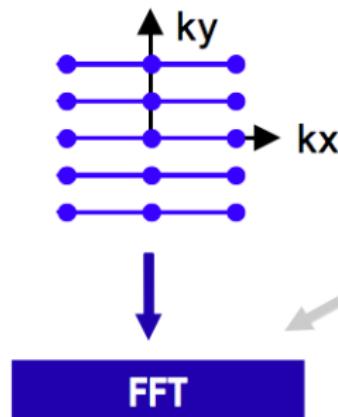
- $W(\mathbf{k})$ is the weighting function to account for nonuniform sampling;
- $s(\mathbf{k})$ is the measured k -space data.

The reconstruction is an inverse fast Fourier Transform on $s(\mathbf{k})$.

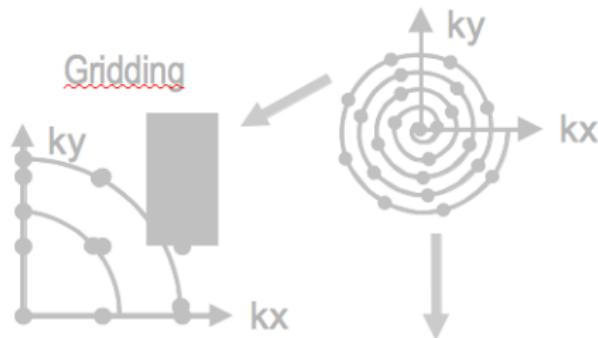
Cartesian trajectory with FFT reconstruction

Reconstructing MR Images

Cartesian Scan Data



Spiral Scan Data

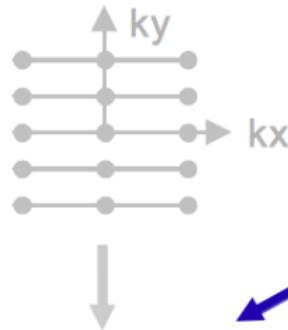


Cartesian scan data + FFT:
Slow scan, fast reconstruction, images may be poor

spiral trajectory, gridding to enable FFT

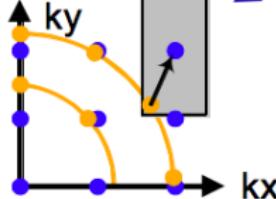
Reconstructing MR Images

Cartesian Scan Data

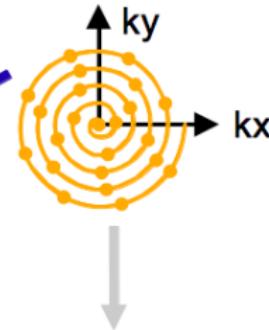


FFT

Gridding¹



Spiral Scan Data



LS

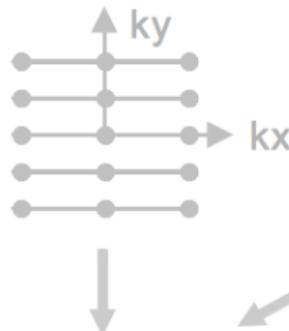
Spiral scan data + Gridding + FFT:
Fast scan, fast reconstruction, better images

¹Based on Fig 1 of Lustig et al, Fast Spiral Fourier Transform for Iterative MR Image Reconstruction, IEEE Int'l Symp. on Biomedical Imaging, 2004

spiral trajectory with linear solver reconstruction

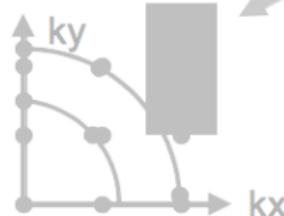
Reconstructing MR Images

Cartesian Scan Data

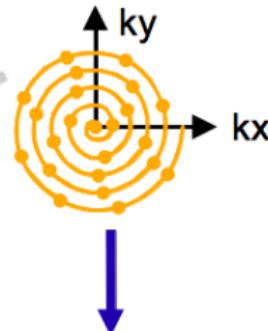


FFT

Gridding



Spiral Scan Data



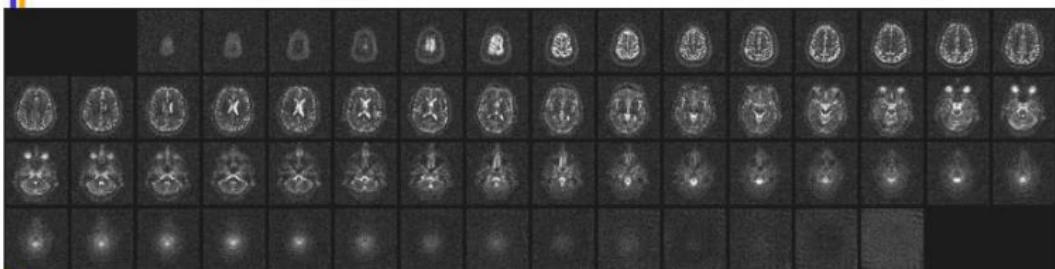
Least-Squares (LS)

Spiral scan data + LS

Superior images at expense of significantly more computation

sodium is much less abundant than water

An Exciting Revolution - Sodium Map of



- Images of sodium in the brain
 - Very large number of samples for increased SNR
 - Requires high-quality reconstruction
- Enables study of brain-cell viability before anatomic changes occur in stroke and cancer treatment – within days!

Courtesy of Keith Thulborn and Ian Atkinson, Center for MR Research, University of Illinois at Chicago

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
University of Illinois, Urbana-Champaign

8

Advanced MRI Reconstruction

1 an Application Case Study

- magnetic resonance imaging
- iterative reconstruction

2 Acceleration on GPU

- determining the kernel parallelism structure
- loop splitting
- loop interchange
- using registers to reduce memory accesses
- chunking data to fit into constant memory
- using hardware trigonometry functions

a linear least squares problem

A quasi-Bayesian estimation problem:

$$\hat{\rho} = \arg \min_{\rho} \underbrace{||\mathbf{F}\rho - \mathbf{d}||_2^2}_{\text{data fidelity}} + \underbrace{||\mathbf{W}\rho||_2^2}_{\text{prior info}},$$

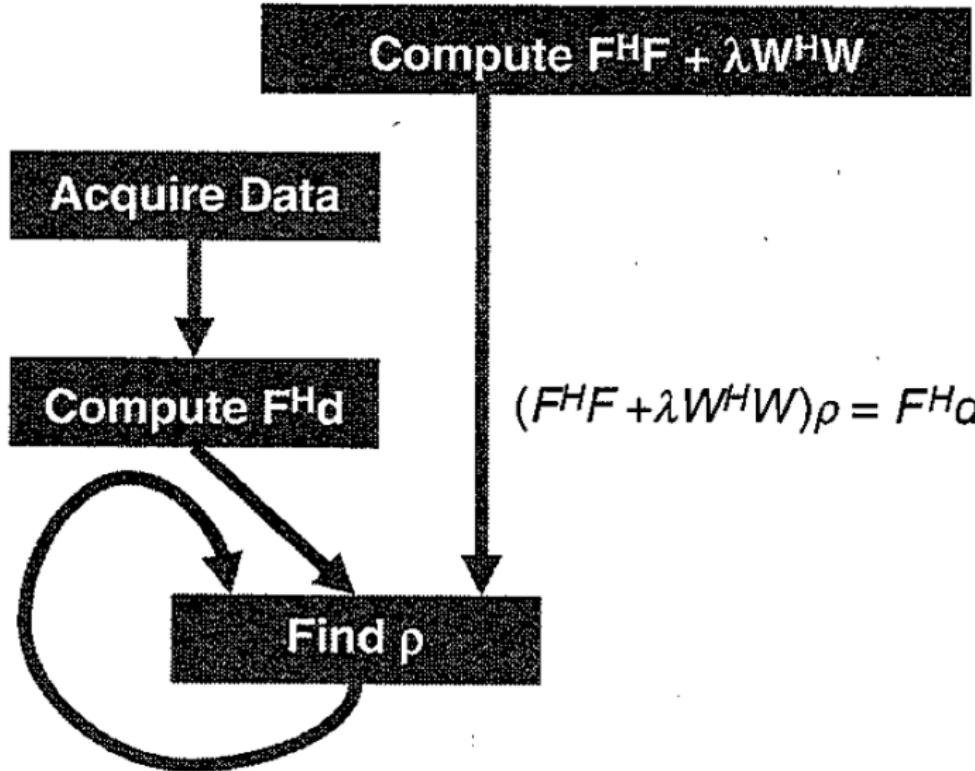
where

- $\hat{\rho}$ contains voxel values for reconstructed image,
- the matrix \mathbf{F} models the imaging process,
- \mathbf{d} is a vector of data samples, and
- the matrix \mathbf{W} incorporates prior information, derived from reference images.

The solution to this linear least squares problem is

$$\hat{\rho} = (\mathbf{F}^H \mathbf{F} + \mathbf{W}^H \mathbf{W})^{-1} \mathbf{F}^H \mathbf{d}.$$

an iterative linear solver



three primary computations

The advanced reconstruction algorithm consists of

①
$$Q(\mathbf{x}_n) = \sum_{m=1}^M |\phi(\mathbf{k}_m)|^2 e^{i2\pi\mathbf{k}_m \cdot \mathbf{x}_n}$$

where $\phi(\cdot)$ is the Fourier transform of the voxel basis function.

②
$$[\mathbf{F}^H \mathbf{d}]_n = \sum_{m=1}^M \phi^*(\mathbf{k}_m) \mathbf{d}(\mathbf{k}_m) e^{i2\pi\mathbf{k}_m \cdot \mathbf{x}_m}$$

- ③ The conjugate gradient solver performs the matrix inversion to solve $(\mathbf{F}^H \mathbf{F} + \mathbf{W}^H \mathbf{W}) \rho = \mathbf{F}^H \mathbf{d}$.

The calculation for $\mathbf{F}^H \mathbf{d}$ is an excellent candidate for acceleration on the GPU because of its substantial data parallelism.

Advanced MRI Reconstruction

1 an Application Case Study

- magnetic resonance imaging
- iterative reconstruction

2 Acceleration on GPU

- determining the kernel parallelism structure
- loop splitting
- loop interchange
- using registers to reduce memory accesses
- chunking data to fit into constant memory
- using hardware trigonometry functions

computing $\mathbf{F}^H \mathbf{d}$

```
for(m = 0; m < M; m++)
{
    rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m];
    iMu[m] = rPhi[m]*iD[m] - iPhi[m]*rD[m];
    for(n = 0; n < N; n++)
    {
        expFHD = 2*PI*(kx[m]*x[n]
                        + ky[m]*y[n]
                        + kz[m]*z[n]);
        cArg = cos(expFHD);
        sArg = sin(expFHD);
        rFHd[n] += rMu[m]*cArg - iMu[m]*sArg;
        iFHd[n] += iMu[m]*cArg + rMu[m]*sArg;
    }
}
```

Consider the Compute to Global Memory Access (CGMA) ratio.

a first version of the kernel

```
__global__ void cmpFHD ( float* rPhi, iPhi, phiMag,
                        kx, ky, kz, x, y, z, rMu, iMu, int N)
{
    int m = blockIdx.x*FHd_THREADS_PER_BLOCK + threadIdx.x;

    rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m];
    iMu[m] = rPhi[m]*iD[m] - iPhi[m]*rD[m];

    for(n = 0; n < N; n++)
    {
        expFHD = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n]);
        carg = cos(expFHD); sArg = sin(expFHD);
        rFHd[n] += rMu[m]*cArg - iMu[m]*sArg;
        iFHd[n] += iMu[m]*cArg + rMu[m]*sArg;
    }
}
```

Advanced MRI Reconstruction

1 an Application Case Study

- magnetic resonance imaging
- iterative reconstruction

2 Acceleration on GPU

- determining the kernel parallelism structure
- **loop splitting**
- loop interchange
- using registers to reduce memory accesses
- chunking data to fit into constant memory
- using hardware trigonometry functions

splitting the outer loop

```
for(m = 0; m < M; m++)
{
    rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m];
    iMu[m] = rPhi[m]*iD[m] - iPhi[m]*rD[m];
}
for(m = 0; m < M; m++)
{
    for(n = 0; n < N; n++)
    {
        expFHD = 2*PI*(kx[m]*x[n]
                        + ky[m]*y[n]
                        + kz[m]*z[n]);
        cArg = cos(expFHD);
        sArg = sin(expFHD);
        rFHd[n] += rMu[m]*cArg - iMu[m]*sArg;
        iFHd[n] += iMu[m]*cArg + rMu[m]*sArg;
    }
}
```

a kernel for the first loop

We convert the first loop into a CUDA kernel:

```
__global__ void cmpMu ( float *rPhi, iPhi, rD, iD, rMu, iMu)
{
    int m = blockIdx * MU_THREADS_PER_BLOCK + threadIdx.x;

    rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m];
    iMu[m] = rPhi[m]*iD[m] - iPhi[m]*rD[m];
}
```

Because M can be very big, we will have many threads.

For example, if $M = 65,536$, with 512 threads per block,
we have $65,536/512 = 128$ blocks.

a kernel for the second loop

```
__global__ void cmpFHd ( float* rPhi, iPhi, PhiMag,
                        kx, ky, kz, x, y, z, rMu, iMu, int N )
{
    int m = blockIdx.x*FHd_THREADS_PER_BLOCK + threadIdx.x;

    for(n = 0; n < N; n++)
    {
        float expFHd = 2*PI*(kx[m]*x[n]+ky[m]*y[n]
                              +kz[m]*z[n]);
        float cArg = cos(expFHd);
        float sArg = sin(expFHd);

        rFHd[n] += rMu[m]*cArg - iMu[m]*sArg;
        iFHd[n] += iMu[m]*cArg + rMu[m]*sArg;
    }
}
```

Advanced MRI Reconstruction

1 an Application Case Study

- magnetic resonance imaging
- iterative reconstruction

2 Acceleration on GPU

- determining the kernel parallelism structure
- loop splitting
- loop interchange**
- using registers to reduce memory accesses
- chunking data to fit into constant memory
- using hardware trigonometry functions

loop interchange

To avoid conflicts between threads,
we interchange the inner and the outer loops:

```
for(m=0; m<M; m++)
{
    for(n=0; n<N; n++)
    {
        expFHD = 2*PI*(kx[m]*x[n]
                        +ky[m]*y[n]
                        +kz[m]*z[n]);
        cArg = cos(expFHD);
        sArg = sin(expFHD);
        rFHd[n] += rMu[m]*cArg
                    - iMu[m]*sArg;
        iFHd[n] += iMu[m]*cArg
                    + rMu[m]*sArg;
    }
}

for(n=0; n<N; n++)
{
    for(m=0; m<M; m++)
    {
        expFHD = 2*PI*(kx[m]*x[n]
                        +ky[m]*y[n]
                        +ky[m]*y[n]);
        cArg = cos(expFHD);
        sArg = sin(expFHD);
        rFHd[n] += rMu[m]*cArg
                    - iMu[m]*sArg;
        rFHd[n] += iMu[m]*cArg
                    + rMu[m]*sArg;
    }
}
```

In the new kernel, the n -th element will be computed by the n -th thread.

a new kernel

```
__global__ void cmpFHD ( float* rPhi, iPhi, phiMag,
                        kx, ky, kz, x, y, z, rMu, iMu, int M )
{
    int n = blockIdx.x*FHD_THREAD_PER_BLOCK + threadIdx.x;

    for(m = 0; m < M; m++)
    {
        float expFHD = 2*PI*(kx[m]*x[n]+ky[m]*y[n]
                             +kz[m]*z[n]);
        float cArg = cos(expFHD);
        float sArg = sin(expFHD);
        rFHD[n] += rMu[m]*cArg - iMu[m]*sArg;
        iFHD[n] += iMu[m]*cArg + rMu[m]*sArg;
    }
}
```

For a 128^3 image, there are $(2^7)^3 = 2,097,152$ threads.

For higher resolutions, e.g.: 512^3 , multiple kernels may be needed.



Advanced MRI Reconstruction

1 an Application Case Study

- magnetic resonance imaging
- iterative reconstruction

2 Acceleration on GPU

- determining the kernel parallelism structure
- loop splitting
- loop interchange
- using registers to reduce memory accesses**
- chunking data to fit into constant memory
- using hardware trigonometry functions

using registers to reduce memory accesses

```
__global__ void cmpFHD ( float* rPhi, iPhi, phiMag,
                        kx, ky, kz, x, y, z, rMu, iMu, int M )
{
    int n = blockIdx.x*FHD_THREAD_PER_BLOCK + threadIdx.x;
    float xn = x[n]; float yn = y[n]; float zn = z[n];
    float rFHdn = rFHd[n]; float iFHdn = iFHd[n];
    for(m = 0; m < M; m++)
    {
        float expFHD = 2*PI*(kx[m]*xn+ky[m]*yn+kz[m]*zn);
        float cArg = cos(expFHD);
        float sArg = sin(expFHD);
        rFHdn += rMu[m]*cArg - iMu[m]*sArg;
        iFHdn += iMu[m]*cArg + rMu[m]*sArg;
    }
    rFHd[n] = rFHdn; iFHd[n] = iFHdn;
}
```

Consider the improved Compute to Memory Access (CGMA) ratio.



Advanced MRI Reconstruction

1 an Application Case Study

- magnetic resonance imaging
- iterative reconstruction

2 Acceleration on GPU

- determining the kernel parallelism structure
- loop splitting
- loop interchange
- using registers to reduce memory accesses
- **chunking data to fit into constant memory**
- using hardware trigonometry functions

chunking k-space data into constant memory

Using constant memory we use cache more efficiently.

Limited in size to 64KB, we need to invoke the kernel multiple times.

```
__constant__ float kx[CHUNK_SZ],ky[CHUNK_SZ],kz[CHUNK_SZ];
// code omitted ...
for(i = 0; k < M/CHUNK_SZ; i++)
{
    cudaMemcpy(kx,&kx[i*CHUNK_SZ],4*CHUNK_SZ,
              cudaMemcpyHostToDevice);
    cudaMemcpy(ky,&ky[i*CHUNK_SZ],4*CHUNK_SZ,
              cudaMemcpyHostToDevice);
    cudaMemcpy(kz,&kz[i*CHUNK_SZ],4*CHUNK_SZ,
              cudaMemcpyHostToDevice);
// code omitted ...
    cmpFHD<<<FHd_THREADS_PER_BLOCK,
                  N/FHd_THREADS_PER_BLOCK>>>
    (rPhi,iPhi,phiMag,x,y,z,rMu,iMu,M);
}
```

adjusting the memory layout

Due to size limitations of constant memory and cache, instead of storing the components of k -space data in three separate arrays, we use an array of structs:

```
struct kdata
{
    float x, float y, float z;
}
__constant struct kdata k[CHUNK_SZ];
```

and then in the kernel we use `k[m].x`, `k[m].y`, and `k[m].z`.

Advanced MRI Reconstruction

1 an Application Case Study

- magnetic resonance imaging
- iterative reconstruction

2 Acceleration on GPU

- determining the kernel parallelism structure
- loop splitting
- loop interchange
- using registers to reduce memory accesses
- chunking data to fit into constant memory
- **using hardware trigonometry functions**

using hardware trigonometry functions

Instead of `cos` and `sin` as implemented in software, the hardware versions `_cos` and `_sin` provide a much higher throughput.

The `_cos` and `_sin` are implemented as hardware instructions executed by the special function units.

We need to be careful about a loss of accuracy.

The validation involves a “perfect” image:

- a reverse process to generate “scanned” data;
- metrics: mean square error & signal-to-noise ratios.

The last stage is the experimental performance tuning.

references

This lecture is based on Chapter 8 (first edition; or Chapter 11 for the second edition) in the book of Kirk & Hwu.

- A. Lu, I.C. Atkinson, and K.R. Thulborn. **Sodium Magnetic Resonance Imaging and its Bioscale of Tissue Sodium Concentration**. *Encyclopedia of Magnetic Resonance*, John Wiley & Sons, 2010.
- S.S. Stone, J.P. Haldar, S.C. Tsao, W.-m.W. Hwu, B.P. Sutton, and Z.-P. Liang. **Accelerating advanced MRI reconstructions on GPUs**. *Journal of Parallel and Distributed Computing* 68(10): 1307–1318, 2008.
- The IMPATIENT MRI Toolset, open source software available at <http://impact.crhc.illinois.edu/mri.php>.

Disk Based Parallelism

1 Disk Based Parallelism

- when random access memory is not large enough
- an example: the pancake sorting problem

2 Roomy: A System for Space Limited Computations

- extending C/C++ with a software library
- programming with Roomy

3 Big Data

- Hadoop and the Map/Reduce model

MCS 572 Lecture 23
Introduction to Supercomputing
Jan Verschelde, 14 October 2016

Disk Based Parallelism

1 Disk Based Parallelism

- when random access memory is not large enough
- an example: the pancake sorting problem

2 Roomy: A System for Space Limited Computations

- extending C/C++ with a software library
- programming with Roomy

3 Big Data

- Hadoop and the Map/Reduce model

processing large volumes of data

For the context of this lecture, consider:

- big data applications

In applications with huge volumes of data,
the bottleneck is not the number of arithmetical operations.

- the disk is the new RAM

`Roomy` is a software library that allows to treat disk
as Random Access Memory.

An application of `Roomy` concerns the determination
of the minimal number of moves to solve Rubik's cube.

Parallel Disk-based Computation

Parallel disk-based computation: instead of Random Access Memory, use disks as the main working memory of a computation. This gives much more space for the same price.

Performance issues and solutions:

- Bandwidth: the bandwidth of a disk is roughly 50 times less than RAM (100 MB/s versus 5 GB/s).

Solution: use many disks in parallel.

- Latency: even worse, the latency of disk is many orders of magnitude worse than RAM.

Solution: avoid latency penalties by using streaming access.

Disk Based Parallelism

1 Disk Based Parallelism

- when random access memory is not large enough
- an example: the pancake sorting problem

2 Roomy: A System for Space Limited Computations

- extending C/C++ with a software library
- programming with Roomy

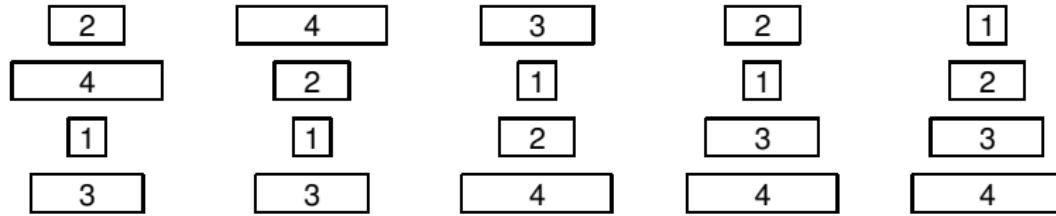
3 Big Data

- Hadoop and the Map/Reduce model

sorting pancakes

Given a stack of n numbered pancakes.

A spatula can reverse the order of the top k pancakes for $2 \leq k \leq n$.



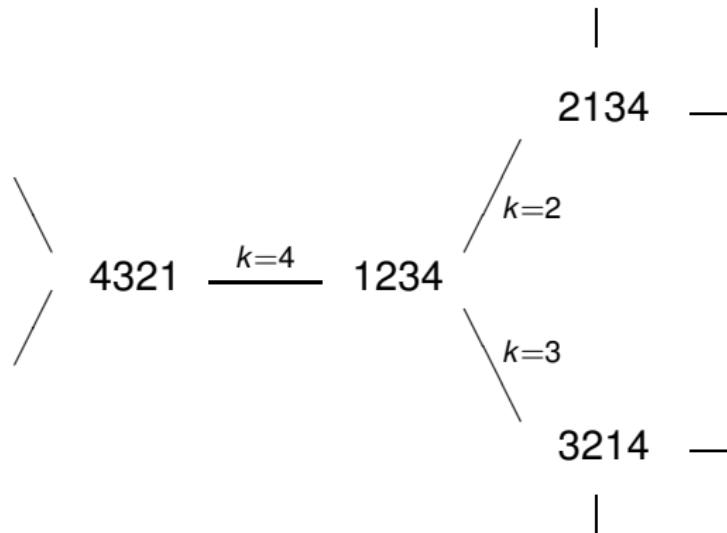
The sort happened with 4 flips:

- 1 k = 2
- 2 k = 4
- 3 k = 3
- 4 k = 2

How many flips (prefix reversals) are sufficient to sort?

pancake sorting graph

We generate all permutations using the flips:



There are 3 stacks that require 1 flip.

$4! = 24$ permutations, $24 = 1 + 3 + 6 + 11 + 3$,
at most 4 flips are needed for a stack of 4 pancakes.

breadth first search, running pancake

```
$ mpiexec -np 2 ./pancake
Sun Mar  4 13:24:44 2012: BEGINNING 11 PANCAKE BFS
Sun Mar  4 13:24:44 2012: Initial one-bit RoomyArray constructed
Sun Mar  4 13:24:44 2012: Level 0 done: 1 elements
Sun Mar  4 13:24:44 2012: Level 1 done: 10 elements
Sun Mar  4 13:24:44 2012: Level 2 done: 90 elements
Sun Mar  4 13:24:44 2012: Level 3 done: 809 elements
Sun Mar  4 13:24:44 2012: Level 4 done: 6429 elements
Sun Mar  4 13:24:44 2012: Level 5 done: 43891 elements
Sun Mar  4 13:24:45 2012: Level 6 done: 252737 elements
Sun Mar  4 13:24:49 2012: Level 7 done: 1174766 elements
Sun Mar  4 13:25:04 2012: Level 8 done: 4126515 elements
Sun Mar  4 13:25:45 2012: Level 9 done: 9981073 elements
Sun Mar  4 13:26:48 2012: Level 10 done: 14250471 elements
Sun Mar  4 13:27:37 2012: Level 11 done: 9123648 elements
Sun Mar  4 13:27:53 2012: Level 12 done: 956354 elements
Sun Mar  4 13:27:54 2012: Level 13 done: 6 elements
Sun Mar  4 13:27:54 2012: Level 14 done: 0 elements
Sun Mar  4 13:27:54 2012: ONE-BIT PANCAKE BFS DONE
Sun Mar  4 13:27:54 2012: Elapsed time: 3 m, 10 s, 536 ms, 720 us
```

how many flips to sort a stack of 11 pancakes?

$$11 \times 10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 = 39,916,800$$

| #moves | #stacks |
|--------|----------|
| 0 | 1 |
| 1 | 10 |
| 2 | 90 |
| 3 | 809 |
| 4 | 6429 |
| 5 | 43891 |
| 6 | 252737 |
| 7 | 1174766 |
| 8 | 4126515 |
| 9 | 9981073 |
| 10 | 14250471 |
| 11 | 9123648 |
| 12 | 956354 |
| 13 | 6 |
| total | 39916800 |

⇒ at most 13 moves for a stack of 11 pancakes.

Disk Based Parallelism

1 Disk Based Parallelism

- when random access memory is not large enough
- an example: the pancake sorting problem

2 Roomy: A System for Space Limited Computations

- extending C/C++ with a software library
- programming with Roomy

3 Big Data

- Hadoop and the Map/Reduce model

Roomy

Roomy is

- a new programming model that extends a programming language with transparent disk-based computing supports;
- an open source C/C++ library implementing this new programming language extension;
- available at [sourceforge](#);
- written by Daniel Kunkle in the lab of Gene Cooperman.

Applied to problems in computational group theory, and large enumerations, e.g.: Rubik's cube can be solved in 26 moves or less.

disk is the new RAM

Example: 50-node cluster with 200GB disk space per computer gives 10TB per computer. Bandwidth of one disk: 100MB/s, bandwidth of 50 disks in parallel: 5GB/s

⇒ The 10TB disk space is considered as RAM.

Some caveats:

- Disk latency remains limiting.
Use “old-fashioned” RAM as cache.
If disk is the new RAM, RAM is the new cache.
- Networks must restructured to emphasize local access over network access.

the Roomy programming model

The Roomy programming programming model

- provides basic data structures: arrays, lists, and hash tables;
- transparently distributes data structures across many disks and performs operations on that data in parallel;
- immediately process streaming access operators;
- delays processing random operators until they can be performed efficiently in batch.

For example: collecting and sorting updates to an array.

Disk Based Parallelism

1 Disk Based Parallelism

- when random access memory is not large enough
- an example: the pancake sorting problem

2 Roomy: A System for Space Limited Computations

- extending C/C++ with a software library
- programming with Roomy

3 Big Data

- Hadoop and the Map/Reduce model

programming construct: map

```
RoomyArray* ra;
RoomyList* rl;
// Function to map over ra.
void mapFunc ( uint64 i, void* val )
{
    RoomyListadd(rl,val);
}

int main ( int argc, char **argv )
{
    Roomy_init(&argc,&argv);
    ra = RoomyArray_makeBytes("array",sizeof(uint64),100);
    rl = RoomyList_make("list",sizeof(uint64));
    RoomyArray_map(ra,mapFunc); // execute map
    RoomyList_sync(rl); // synchronize list for delayed ads
    Roomy_finalize();
}
```

programming construct: reduce

Computing the sum of squares of the elements in a RoomyList:

```
RoomyList* rl; // elements of type int

// add square of an element to sum
void mergeElt ( int* sum, int* element )
{
    *sum += (*e) * (*e);
}

// compute sum of two partial answers
void mergeResults ( int * sum1, int* sum2 )
{
    *sum1 += *sum2;
}
int sum = 0;
RoomyList_reduce(rl,&sum,sizeof(int),
                 mergeElt,mergeResults);
```

programming construct: predicate

Predicates count the number of elements in a data structure that satisfy a Boolean function.

```
RoomyList* rl;
```

```
// predicate: return 1 if element is greater than 42
uint8 predFunc ( int* val )
{
    return (*val > 42) ? 1 : 0;
}
```

```
RoomyList_attachPredicate(rl,predFunc);
```

```
uint64 gt42 = RoomyList_predicateCount(rl,predFunc);
```

programming construct: permutation multiplication

For permutations X, Y, Z on N elements length N do Z = X*Y as
for i=0 to N-1: $Z[i] = Y[X[i]]$.

```
RoomyArray *X, *Y, *Z;
// access X[i]
void accessX ( unit64 i, uint64* x_i) {
    RoomyArray_access(Y, *x_i,&i, accessY);
}
// access Y[X[i]]
void accessY ( uint64 x_i, uint64* y_x_i, uint64* i )
{
    RoomyArray_update(Z,*i,y_x_i, setZ);
}
// set Z[i] = Y[X[i]]
void setZ ( uint64, i, uint64* z_i, uint64* y_x_i, uint64* z_i_NEW )
{
    *z_i_NEW = *y_x_i;
}
RoomyArray_map(X,accessX); // access X[i]
RoomyArray_sync(Y);           // access Y[X[i]]
RoomyArray_sync(Z);           // set Z[i] = Y[X[i]]
```

programming construct: set operations

Convert list to set:

```
RoomyList *A; // can contain duplicates  
RoomyList_removeDuplicates(A); // is a set
```

Union of two sets $A \cup B$:

```
RoomyList *A, *B;  
RoomyList_addAll(A, B);  
RoomyList_removeDuplicates(A);
```

Difference of two sets $A \setminus B$:

```
RoomyList *A, *B;  
RoomyList_removeAll(A, B);
```

The intersection $A \cap B$ is implemented as $(A \cup B) \setminus (A \setminus B) \setminus (B \setminus A)$.

programming construct: breadth-first search

Initialize the search:

```
// Lists of all elements, current, and next level
RoomyList* all = RoomyList_make("allLev",eltSize);
RoomyList* cur = RoomyList_make("lev0",eltSize);
RoomyList* next = RoomyList_Make("lev1",eltSize);

// Function to produce next level from current
void genNext ( T elt )
{
    // user defined code to compute neighbors ...
    for nbr in neighbors
        RoomyList_add(next,nbr);
}
// add start element
RoomyList_add(all,startElt);
RoomyList_add(cur,startElt);
```

perform the search

```
// generate levels until no new states are found
while(RoomyList_size(cur))
{ // generate next level from current
    RoomyList_map(cur,genNext);
    RoomyList_sync(next);
    // detect duplicates within next level
    RoomyList_removeDups(next);
    // detect duplicates from previous levels
    RoomyList_removeAll(next,all);
    // record new elements
    RoomyList_addAll(all,next);
    // rotate levels
    RoomyList_destroy(cur);
    cur = next;
    newt = RoomyList_make(levName,eltSize);
}
```

Disk Based Parallelism

1 Disk Based Parallelism

- when random access memory is not large enough
- an example: the pancake sorting problem

2 Roomy: A System for Space Limited Computations

- extending C/C++ with a software library
- programming with Roomy

3 Big Data

- Hadoop and the Map/Reduce model

Large Data Files

Process big data with parallel and distributed computing:

- MySQL scales well.
- overview of Hadoop:

Hadoop has become the de facto standard for processing big data.

Hadoop is used by Facebook to store photos, by LinkedIn to generate recommendations, and by Amazon to generate search indices.

Hadoop works by connecting many different computers, hiding the complexity from the user: work with one giant computer.

Hadoop uses a model called Map/Reduce.

RHadoop by Antonio Piccolboni is a project for R and Hadoop, hosted at <https://github.com>.

the Map/Reduce model

Goal: help with writing efficient parallel programs.

Common data processing tasks: filtering, merging, aggregating data and many (not all) machine learning algorithms fit into Map/Reduce.

Two steps:

Map: Tasks read in input data as a set of records, process the records, and send groups of similar records to reducers.

The mapper extracts a key from each input record.

Hadoop will then route all records with the same key to the same reducer.

Reduce: Tasks read in a set of related records, process the records, and write the results.

The reducer iterates through all results for the same key, processing the data and writing out the results.

Strength: the map and reduce steps run well in parallel.

example: predicting user behavior

Problem: how likely is a user to purchase an item from a website?
Suppose we have already computed (maybe using Map/Reduce)
a set of variables describing each user:

- most common locations,
- the number of pages viewed,
- the number of purchases made in the past.

Calculate forecast using random forests:

- Random forests work by calculating a set of regression trees and then averaging them together to create a single model.
- It can be time consuming to fit the random trees to the data, but each new tree can be calculated independently.
- One way to tackle this problem is to use a set of map tasks to generate random trees, and then send the models to a single reducer task to average the results and produce the model.

suggested reading

- Daniel Kunkle. **Roomy: A C/C++ library for parallel disk-based computation, 2010.** <http://roomy.sourceforge.net/>.
- Daniel Kunkle and Gene Cooperman.
Harnessing parallel disks to solve Rubik's cube.
Journal of Symbolic Computation 44:872-890, 2009.
- Daniel Kunkle and Gene Cooperman.
Solving Rubik's Cube: Disk is the new RAM.
Communications of the ACM 51(4):31-33, 2008.
- Garry Turkington. Hadoop Beginner's Guide.
www.it-ebooks.info.

Summary + Exercises

With Roomy we can solve problems that would exhaust RAM.
Computational group theory often requires enumeration.

Exercises:

- ① Watch the YouTube google tech talk of Gene Cooperman on the application of disk parallelism to Rubik's cube.
- ② Read the paper of Daniel Kunkle and Gene Cooperman that was published in the Journal of Symbolic Computation.

The midterm exam of Friday can be in class or take home.

You can postpone the decision till 12:50PM on Friday 21 October.

Introduction to Hadoop

1 What is Hadoop?

- the big data revolution
- extracting value from data
- cloud computing

2 Understanding MapReduce

- the word count problem
- more examples

MCS 572 Lecture 24
Introduction to Supercomputing
Jan Verschelde, 17 October 2016

Introduction to Hadoop

1 What is Hadoop?

- the big data revolution
- extracting value from data
- cloud computing

2 Understanding MapReduce

- the word count problem
- more examples

extract meaning from the data avalanche

goal: applying complex analytics to large data sets

complementary technology is the use of cloud computing,
in particular: Amazon Web Services.

Assumptions for writing MapReduce applications:

- ① comfortable writing Java programs; and
- ② familiar with the Unix command-line interface.

What is the value of data?

- Some questions are relevant only for large data sets.
For example: movie preferences are inaccurate when based on just another person, but patterns can be extracted from the viewing history of millions.
- Big data tools enable processing on larger scale at lower cost.
- Additional hardware is needed to make up for latency.
- The notion of what is a database should be revisited.

Realize that one no longer needs to be among the largest corporations or government agencies to extract value from data.

data processing systems

Two ways to process large data sets:

- ① *scale-up*: large and expensive computer (supercomputer).
We move the same software onto larger and larger servers.
- ② *scale-out*: spread processing onto more and more machines (commodity cluster).

Limiting factors:

- complexity of concurrency in multiple CPUs;
- CPUs are much faster than memory and hard disk speeds.

There is a third way:

- ③ *Cloud computing*: provider deals with scaling problems.

Introduction to Hadoop

1 What is Hadoop?

- the big data revolution
- **extracting value from data**
- cloud computing

2 Understanding MapReduce

- the word count problem
- more examples

principles of Hadoop

- ① *All roads lead to scale-out*, scale-up architectures are rarely used and scale-out is the standard in big data processing.
- ② *Share nothing*: communication and dependencies are bottlenecks, individual components should be as independent as possible to allow to proceed regardless of whether others fail.
- ③ *Expect failure*: components will fail at inconvenient times.
See our previous exercise on multi-component expected life span; resilient scale-up architectures require much effort.
- ④ *Smart software, dumb hardware*: push smarts in the software, responsible for allocating generic hardware.
- ⑤ *Move processing, not data*: perform processing locally on data.
What gets moved through the network are program binaries and status reports, which are dwarfed in size by the actual data set.
- ⑥ *Build applications, not infrastructure*.
Instead of placing focus on data movement and processing, work on job scheduling, error handling, and coordination.

History of Hadoop

Who to thank for Hadoop?

- ① Google released two academic papers on their technology: in 2003: the google file system; and in 2004: MapReduce. The papers are available for download from google.research.com.
- ② Doug Cutting started implementing Google systems, as a project within the Apache open source foundation.
- ③ Yahoo hired Doug Cutting in 2006 and supported Hadoop project.

From Tom White: *Hadoop: The Definitive Guide*, published by O'Reilly:

Definition (The name Hadoop is ...)

The name my kid gave a stuffed yellow elephant. Short, relatively easy to spell and pronounce, meaningless, and not used elsewhere: those are my naming criteria. Kids are good at generating such.

components of Hadoop

Two main components of Hadoop are

- ① The Hadoop Distributed File System (HDFS)
stores very large data sets across a cluster of hosts,
 - ▶ optimized for throughput instead of latency,
 - ▶ achieving high availability through replication instead of redundancy.
- ② MapReduce is a data processing paradigm that takes a specification of input (map) and output (reduce) and applies this to the data.

MapReduce integrates tightly with HDFS,
running directly on HDFS nodes.

common building blocks

- run on commodity clusters, scale-out: can add more servers
- have mechanisms for identifying and working around failures
- provides services transparently, user can concentrate on problem
- software cluster sitting on physical server controls execution

the Hadoop Distributed File System (HDFS)

HDFS spreads the storage across nodes.

Features:

- files stored in blocks of 64MB (typical file system: 4-32 KB)
- optimized for throughput over latency, efficient at streaming, but poor at seek requests for many small ones
- optimized for workloads that are read-many and write-once
- storage node runs process DataNode that manages blocks, coordinated by master NameNode process running on separate host
- replicates blocks on multiple nodes to handle disk failures

MapReduce

A new technology build on fundamental concepts:

- functional programming languages offer map and reduce;
- divide and conquer breaks problem into multiple subtasks.

The input data goes to a series of transformations:

- the developer defines the data transformations,
- Hadoop's MapReduce job manages parallel execution.

What is the point?

Unlike relational databases the required structure data, the data is provided as a series of key-value pairs and the output of the map is another set of key-value pairs.

Most important point: *the Hadoop platform takes responsibility for every aspect of executing the processing across the data.*

The user is unaware of the actual size of data and cluster, the platform determines how best to utilize all the hosts.

Challenge to the user: break the problem into

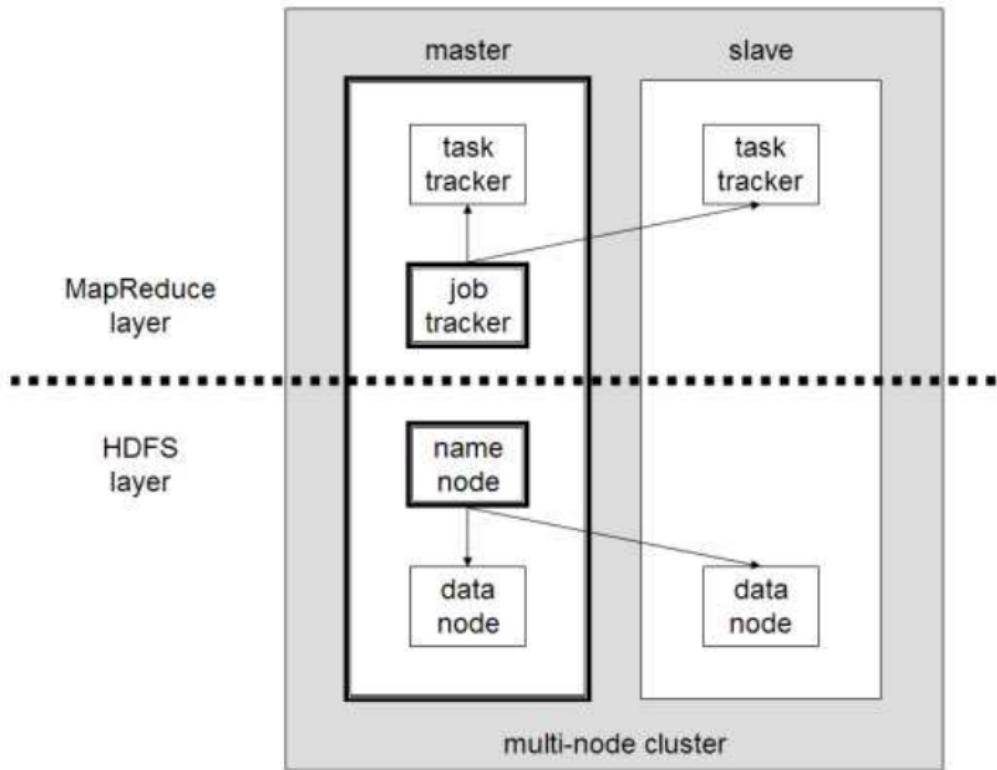
- ① the best combination of chains of map and reduce functions, where the output of one is the input of the next; and
- ② where each chain could be applied independently to each data element, allowing for parallelism.

software clusters

The common architecture of HDFS and MapReduce are software clusters:

- cluster of worker nodes managed by a coordinator node;
- master (NameNode for HDFS and JobTracker for MapReduce) monitors clusters and handles failures;
- processes on server (DataNode for HDFS and TaskTracker for MapReduce) perform work on physical host, receiving instructions for master and reporting status.

a multi-node Hadoop cluster



copied from wikipedia

what is it good for, and what not

Strengths and weaknesses:

- + Hadoop is flexible and scalable data processing platform;
- batch processing not for real time, e.g.: serving web queries.

Application by Google:

- ① Web crawler retrieves updated webpage data.
- ② MapReduces processes the huge data set.
- ③ The web index is then used by a fleet of MySQL servers for search requests.

Introduction to Hadoop

1 What is Hadoop?

- the big data revolution
- extracting value from data
- cloud computing

2 Understanding MapReduce

- the word count problem
- more examples

cloud computing

Cloud computing with Amazon Web Services is another technology.

Two main aspects:

- ① new architecture option; and
- ② different approach to cost.

Instead of running your own clusters, all you need is a credit card.

This is the third way:

- ① scale-up: supercomputer
- ② scale-out: commodity cluster
- ③ cloud computing: the provider deals with the scaling problem.

using cloud computing

How does it work? Two steps:

- ① user develops software according to published guidelines or interface;
- ② deploy onto the cloud platform and allow the service to scale on demand.

Service-on-demand aspect:

- start application on small hardware and scale up,
- off loading infrastructure costs to cloud provider.

Major benefit: great reduction of cost of entry for organization to launch an online service.

Amazon Web Services (AWS)

AWS is an infrastructure on demand

A set of cloud computing services:

- Elastic Compute Cloud (EC2): a server on demand.
- Simple Storage Service (S3): programmatic interface to store objects.
- Elastic MapReduce (EMR): Hadoop in the cloud.

In most impressive mode: EMR pulls data from S3, process on EC2.

Introduction to Hadoop

1 What is Hadoop?

- the big data revolution
- extracting value from data
- cloud computing

2 Understanding MapReduce

- the word count problem
- more examples

key/value transformations

MapReduce as a series of key/value transformations

```
map           reduce  
{K1, V1} --> {K2, List<V2>} -----> {K3, V3}
```

Two steps, three sets of key-value pairs:

- ➊ The input to the map is a series of key-value pairs, called K1 and V1.
- ➋ The output of the map (and the input to the reduce) is a series of keys and an associated list of values that are called K2 and V2.
- ➌ The output of the reduce is another series of key-value pairs, called K3 and V3.

Hadoop's hello world

The word count problem:

- Input: a text file.
- Output: the frequency of words.

Project Gutenberg (at www.gutenberg.org) offers many free books (mostly classics) in plain text file format.

Doing a word count is a very basic data analytic, same authors will use the same patterns of words.

solving the word count problem with MapReduce

Every word on the text file is a key.

Its value is the count, its number of occurrences.

- Keys must be unique, but values need not be.
- Each value must be associated with a key,
but a key could have no values

One must be careful when defining what is a key,
e.g. for the word problem do we distinguish between lower and upper
case words?

a MapReduce job

A complete MapReduce Job for the word count problem:

- 1 Input to the map:

K1/V1 pairs are in the form

< line number, text on the line >.

The mapper takes the line and breaks it up into words.

- 2 Output of the map, input of reduce:

K2/V2 pairs are in the form < word, 1 >.

- 3 Output of reduce:

K3/V3 pairs are in the form < word, count >.

pseudo code for the word count problem

```
map(String key, String value):  
    // key: document name  
    // value: document contents  
    for each word w in value:  
        EmitIntermediate(w, "1");  
  
reduce(String key, Iterator values):  
    // key: a word  
    // values: a list of counts  
    int result = 0;  
    for each v in values:  
        result += ParseInt(v);  
    Emit(AsString(result));
```

Introduction to Hadoop

1 What is Hadoop?

- the big data revolution
- extracting value from data
- cloud computing

2 Understanding MapReduce

- the word count problem
- more examples

grep and counting URLs

Distributed Grep:

- The map function emits a line if it matches a supplied pattern.
- The reduce function is an identity function that just copies the supplied intermediate data to the output.

Count of URL Access Frequency:

- The map function processes logs of web page requests and outputs (URL, 1).
- The reduce function adds together all values for the same URL and emits a (URL, total count) pair.

reverse web-link graph

Reverse Web-Link Graph:

- The map function outputs $(\text{target}, \text{source})$ pairs for each link to a target URL found in a page named `source`.
- The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: $(\text{target}, \text{list}(\text{source}))$.

term-vector per host

Term-Vector per Host:

A term vector summarizes the most important words that occur in a document or a set of documents as a list of (word, frequency) pairs.

- The map function emits a (hostname, term vector) pair for each input document (where the hostname is extracted from the URL of the document).
- The reduce function is passed all per-document term vectors for a given host.

It adds these term vectors together, throwing away infrequent terms, and then emits a final (hostname, term vector) pair.

inverted index

Inverted Index:

- The map function parses each document, and emits a sequence of $(\text{word}, \text{document ID})$ pairs.
- The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a $(\text{word}, \text{list}(\text{document ID}))$ pair.

The set of all output pairs forms a simple inverted index.

It is easy to augment this computation to keep track of word positions.

distributed sort

Distributed Sort:

- The map function extracts the key from each record, and emits a `key, record` pair.
- The reduce function emits all pairs unchanged.

This computation depends

- on the partitioning facilities: hashing keys; and
- and the ordering properties: within a partition key-value pairs are processed in increasing key order.

suggested reading

- Garry Turkington. Hadoop Beginner's Guide.
www.it-ebooks.info.
- Papers available from research.google.com/archive (URL):
 - ▶ Luiz Barroso, Jeffrey Dean, and Urs Hölzle:
Web Search for a Planet: The Google Cluster Architecture.
At URL/googlecluster.html.
 - ▶ Jeffrey Dean and Sanjay Ghemawat:
MapReduce: Simplified Data Processing on Large Clusters.
At URL/mapreduce.html.
Appeared in OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December 2004.
 - ▶ Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung:
The Google File System. At URL/gfs.html.
Appeared in 19th ACM Symposium on Operating Systems Principles, Lake George, NY, October, 2003.

Summary and Exercises

Hadoop and Cloud Computing reduce the entry cost for large scale data processing.

Exercises:

- 1 Install Hadoop on your computer and run the word count problem.
- 2 Consider the pancake problem we have seen in Lecture 23 and formulate a solution within the MapReduce programming model.

Applying Hadoop to a nontrivial problem could be a topic for an interesting final project.