INTRODUCTION TO GPU COMPUTING IN AALTO

Topi Siro
11.6.2014
OUTLINE

• PART I
  • Introduction to GPUs
  • Basics of CUDA (and OpenACC)
  • Running GPU jobs on Triton

• Hands-on I

• PART II
  • Optimizing CUDA codes
  • Libraries

• Hands-on II

• PART III
  • Profiling
PART I
INTRODUCTION TO GPU COMPUTING AND CUDA
INTRODUCTION

• Modern graphics processors evolved from 3D accelerators of the 80s and 90s that were needed to run computer games with increasingly demanding graphics

• Even today, the development is driven by the gaming industry, but general purpose applications are becoming more and more important

• In recent years, GPU computing in various disciplines of science has seen a rapid growth

• The two big players are ATI (AMD) and NVIDIA
  • So far, the majority of general purpose GPU computations have been implemented in NVIDIA’s CUDA framework
The evolution of the theoretical performance of CPUs and GPUs
CPU VS GPU

• In a nutshell, the CPU typically runs one or a few very fast computational threads, while the GPU runs on the order of ten thousand simultaneous (not so fast) threads.

• The hardware in the GPU emphasizes fast data processing over cache and flow control.

• To benefit from the GPU, the computation has to be data-parallel and arithmetically intensive.
  • data-parallel: the same instructions are executed in parallel to a large number of independent data elements.
  • arithmetic intensity: the ratio of arithmetic operations to memory operations.
GPU PROGRAMMING

• **High level**
  • Libraries
    • Linear algebra
    • FFT
    • Thrust (STL for CUDA)
    • etc.
  • OpenACC
    • OpenMP for GPUs

• **Low level**
  • Write you own kernels
    • CUDA for NVIDIA
    • OpenCL for AMD
GPU PROGRAMMING

• When to use GPUs?
  • The best case scenario is a large data-parallel problem that can be parallelized to tens of thousands of simultaneous threads
  • Typical examples include manipulating large matrices and vectors
  • It might not be worth it if the data is used only once, because the speedup has to overcome the cost of data transfers to and from the GPU

• With an abundance of libraries for standard algorithms, one can in many cases use GPUs in a high-level code without having to know about the details (don’t have to write any kernels)

• However, optimal performance often requires code that is tailor-made for the specific problem

• Some things to consider in low-level GPU programming:
  • How to parallelize?
  • Memory transfers between the host system and the GPU
  • Efficient global memory access within the GPU
  • Shared memory usage
OPENACC

• A directive based approach to GPU computing

• Similar syntax to OpenMP

**Pros:**

• Potentially a fast and easy way to port an existing large code to GPUs

• No need to worry about low level details like thread configuration

**Cons:**

• Still quite young, so still some kinks left to be ironed out

• Not as much control → might not get the best performance
**OPENACC**

**OpenMP:**

```c
void daxpy(int n, double a,
    const double *restrict x,
    double *restrict y)
{
    #pragma omp parallel for
    for (int j=0; j<n; ++j)
        y[j] += a * x[j];
}
```

**OpenACC:**

```c
void daxpy(int n, double a,
    const double *restrict x,
    double *restrict y)
{
    #pragma acc parallel loop present(x,y)
    for (int j=0; j<n; ++j)
        y[j] += a * x[j];
}
```
CUDA

- **CUDA** stands for Compute Unified Device Architecture and is the parallel programming model by NVIDIA for its GPUs.

- The GPUs are programmed with an extension of the C language.

- The major difference compared to ordinary C is the use of special functions, called **kernels**, that are executed on the GPU by many threads in parallel.

- Each thread is assigned an ID number that allows different threads to operate on different data.

- The typical structure of a CUDA program includes:
  1. Prepare data
  2. Allocate memory on the GPU
  3. Transfer data to GPU
  4. Perform the computation on the GPU
  5. Transfer results back to the host side
The GPU consists of multiple streaming multiprocessors (SM) that each have tens or hundreds of cores.

- Each SM has an L1 cache / shared memory.
- There is also an L2 cache shared by all SMs.
Thread hierarchy in CUDA

- The **threads** are organized into **blocks** that form a **grid**
- Threads within a block are executed on the same multiprocessor and they can communicate via the shared memory
- Threads in the same block can be synchronized but different blocks are completely independent
A simple example of a kernel that copies the vector A into vector B with one block of N threads. The kernel is launched from the host code just like any ordinary function, except that the grid configuration is specified in the angled brackets as <<<number of blocks, threads per block>>>.

In the kernel, defined by the __global__ qualifier, the thread number is saved in the variable id. Then, the element in A corresponding to id is copied to the same place in B. This way, thread number 0 copies the element 0, thread number 1 copies the element 1, etc., until N, all at the same time.
• The maximum number of threads per block in current GPUs is 1024

• The thread ID in the variable `threadIdx` can be up to 3-dimensional for convenience
  - `dim3 threadIdx(8, 8);  //2-dimensional block`

• The grid can be 1- or 2-dimensional, and the maximum number of blocks per dimension is 65535 (Fermi) or ~2 billion (Kepler)

• In addition to the thread ID, there are intrinsic variables for the grid dimensions, block ID and the threads per block available in the kernel, called `gridDim`, `blockIdx` and `blockDim`

• The number of threads per block should always be divisible by 32, because threads are executed in groups of 32, called `warps`
A better version of the vector copy kernel for vectors of arbitrary length

Each thread calculates its global ID number

Removing the if statement would result in array overflow, unless N happens to be divisible by blockDim, such that we could launch exactly N threads.
ALLOCATION AND TRANSFER

• The GPU memory is separate from the host memory that the CPU sees

• Before any kernels can be run, the necessary data has to transferred to the GPU memory via the PCIe bus

• Memory on the GPU can be allocated with `cudaMalloc`
  • `cudaMalloc((void**)&d_A, Nbytes,);`

• Memory transfers can be issued with `cudaMemcpy`
  • `cudaMemcpy(d_A, A, Nbytes, cudaMemcpyHostToDevice)`
  • Be careful with the order of arguments (to, from,...)
  • The last argument can be any of cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice or cudaMemcpyHostToHost
The vector copy code including the host part where we allocate the GPU memory and transfer the data to and from the GPU.
• CUDA programs are compiled and linked with the **nvcc** compiler (available in Triton with module load cuda)

• Files with CUDA code need to have the .cu extension

• The syntax is identical to compiling with gcc, for example

• The only thing to remember is to compile for the appropriate GPU architecture, given with the -arch option (-arch sm_20 for Fermi GPUs)

• To compile and link a simple program in a file main.cu
  • `nvcc -arch sm_20 -o programname main.cu`
GPU MEMORIES

- Efficient use of the GPU requires some low-level knowledge about the hardware, especially the different kinds of memories that are available

**Global memory**
- The main memory of the GPU that can be allocated and manipulated from the host
- Very slow compared to registers and shared memory, so kernels should try to minimize global memory reads and writes
- Cached in L1 and L2 caches in Fermi generation and newer GPUs

**Shared memory**
- A fast user-managed cache that can be used in communicating between threads of the same block
- Ideally, the kernel should load the necessary data from the global memory into shared memory, do the computation, and then write the results back to global memory
- Allocated in the kernel with the `__shared__` qualifier
- Can be configured to 16KB or 48KB per multiprocessor
GPU MEMORIES

• **Registers**
  • Variables that are declared in the kernel are stored into registers that are very fast
  • However, using too much registers might result in register spilling to global memory, which might kill performance

• **Textures**
  • The texture memory offers an alternative, cached access to global memory
  • It’s a bit more complicated to use but might sometimes give better performance
Arrays declared in the kernel might lead to register spilling if the array is not indexed by constants.

Dynamic shared memory allocation is possible by giving the number of bytes in the kernel call as a third parameter in the ```<<< >>>``` but then only one allocation is possible.
PARALLEL REDUCTION

• As a slightly more complicated example, let’s look at an implementation of the prefix sum operation, i.e. calculating the sum of the elements of a vector.

• A serial code would do it like this:

```c
int sum = intAr[0];
for (int i = 1; i < N; i++)
{
    sum = sum + intAr[i];
}
```

• A parallel implementation could look like this:

![Parallel reduction diagram]

Finally, to give a concrete example of a kernel, consider a simple example of copying a given vector A to another vector B. An obvious parallelization scheme is to assign one thread to copy one element. Inside the kernel, every thread has an identification number starting from 0, stored in a variable called `threadIdx`. The threads can then be controlled by making the commands in the kernel dependent on this thread ID. An example code with explanation can be seen in Figure 6.
PARALLEL REDUCTION

First we load the vector into shared memory

Then we iterate according to the figure on the previous slide

__syncthreads() is a synchronization barrier that is needed here to prevent a possible race condition

Finally, the first thread writes the result back to global memory

A parallel prefix sum kernel for $N \leq 1024$
UNIFIED MEMORY IN CUDA 6

• The newest CUDA version, 6.0, introduced a new simplified model to managing memory: unified memory.

• It allows the programmer to use a single pointer to address to data on the CPU or the GPU.

• The system automatically moves the data to where it is needed, so no need for cudaMemcpy().

• To use unified memory, it needs to be allocated with cudaMallocManaged().

• Only works with Kepler devices and newer.

• Best performance still with explicit memory management, but unified memory can speed up coding.
UNIFIED MEMORY EXAMPLE

CPU Code

```c
void sortfile(FILE *fp, int N) {
    char *data;
    data = (char *)malloc(N);
    fread(data, 1, N, fp);
    qsort(data, N, 1, compare);
    use_data(data);
    free(data);
}
```

CUDA 6 Code with Unified Memory

```c
void sortfile(FILE *fp, int N) {
    char *data;
    cudaMallocManaged(&data, N);
    fread(data, 1, N, fp);
    qsort<<<...>>>(data, N, 1, compare);
    cudaDeviceSynchronize();
    use_data(data);
    cudaFree(data);
}
```
SUMMARY OF PART I

• Modern GPUs can be used for general purpose calculations which benefit from large-scale parallelization

• CUDA is the programming model by NVIDIA for its GPUs, and it enables writing code that runs on the GPU with an extension of the C programming language

• CUDA programs consist of host code, run on the CPU, and kernels that are executed on the GPU in parallel by threads whose launch configuration is given in the kernel call

• In the kernels, the threads can be identified with the intrinsic variables threadIdx and blockIdx

• The most important memories are the global memory and the per-multiprocessor shared memory, that can be used for communication between threads of the same block

• CUDA programs are compiled with nvcc
HANDS-ON SESSION I
GPU COMPUTING ON TRITON

- [https://wiki.aalto.fi/display/Triton/GPU+computing](https://wiki.aalto.fi/display/Triton/GPU+computing)

- Triton has 10 GPU nodes with 2 Tesla M2090 GPUs (6GB of memory) in each node

- GPU jobs can be run in the Slurm partition gpu

- Before anything else, load the CUDA module with `module load cuda`
GPU COMPUTING ON TRITON

• GPU jobs can be run straight from the command line or by submitting a job script

• On the frontend, you can use salloc and srun:

```
[sirot1@triton CUDA]$ salloc -t 1:00:00 -p gpu --gres=gpu:1 srun --gres=gpu:1 /share/apps/nvidia/cuda_samples_4.1.28/deviceQuery
salloc: Pending job allocation 4298797
salloc: job 4298797 queued and waiting for resources
```

• Or with a script:

```
[sirot1@triton CUDA]$ sbatch job.sh
Submitted batch job 4298804
[sirot1@triton CUDA]$
```
#!/bin/bash

## name of your job
#SBATCH -J GPUjob

## system message output file
#SBATCH -o gpu_%j.out

## how long a job takes, wallclock time hh:mm:ss
#SBATCH -t 01:01:00

## number of CPU cores
#SBATCH -n 1

## partition: gpu
#SBATCH -p gpu

## how many GPUs per node
#SBATCH --gres=gpu:2

module load cuda/4.1

## run my GPU accelerated executable with --gres
srun --gres=gpu:1 /share/apps/nvidia/cuda_samples_4.1.28/deviceQuery
CUDA Device Query (Runtime API) version (CUDART static linking)

Found 1 CUDA Capable device(s)

Device 0: "Tesla M2090"

- CUDA Driver Version / Runtime Version: 5.5 / 4.1
- CUDA Capability Major/Minor version number: 2.0
- Total amount of global memory: 5375 MBytes (5636554752 bytes)
- (16) Multiprocessors x (32) CUDA Cores/MP: 512 CUDA Cores
- GPU Clock Speed: 1.30 GHz
- Memory Clock rate: 1848.00 Mhz
- Memory Bus Width: 384-bit
- L2 Cache Size: 786432 bytes
- Max Texture Dimension Size (x,y,z): 1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048)
- Max Layered Texture Size (dim) x layers: 1D=(16384) x 2048, 2D=(16384,16384) x 2048
- Total amount of constant memory: 65536 bytes
- Total amount of shared memory per block: 49152 bytes
- Total number of registers available per block: 32768
- Warp size: 32
- Maximum number of threads per block: 1024
- Maximum sizes of each dimension of a block: 1024 x 1024 x 64
- Maximum sizes of each dimension of a grid: 65535 x 65535 x 65535
- Maximum memory pitch: 2147483647 bytes
- Texture alignment: 512 bytes
- Concurrent copy and execution: Yes with 2 copy engine(s)
GPU COMPUTING ON TRITON

• You can check the queue status with `squeue -p gpu`

![squeue output](image)

• You can cancel jobs with `scancel <JOBID>`
HANDS-ON SESSION I

• Log on to Triton
  • `ssh -X <username>@triton.aalto.fi`

• Get the exercises from /triton/scip/GPUHandson.tar
  • `cp /triton/scip/GPUHandson.tar .`

• Extract
  • `tar xvf GPUHandson.tar`

• Now the exercises can be found in the subdirectories of ./GPUHandson

• Each exercise folder contains the .cu file that needs work, a Makefile for easy compiling, a job script for running the code in the queue system and an info.txt that tells you what you need to do

• There is also a Solutions subfolder with the right answer, but no peeking until you’ve tried your best!
HANDS-ON SESSION 1

• Edit the .cu files with your favorite text editor
  • I only know about emacs. Turn on c++ highlighting by **Esc-x-c++-Enter**

• Compile the code with **make**

• Run the program with **sbatch job.sh**
  • The program will be submitted to the gpu queue. We have a couple of nodes reserved and the jobs should be very short but you might want to check the queue status with **squeue -p gpu**
  • The output will be written in a `<excercise_name>.out` file
PART II
MAXIMIZING PERFORMANCE
COALESCED MEMORY ACCESS

- The memory access pattern of the threads can have dramatic consequences for the memory bandwidth

- In Fermi and newer GPUs, the memory can be only read in 128 byte chunks

- In the best case scenario, the nth thread accesses the nth word in the memory, and the access is aligned so that the starting address is a multiple of 128
COALESCED MEMORY ACCESS

- In newer GPUs the requirements for coalescing are much more relaxed.

- Still, storing and accessing data inefficiently may cripple your performance.

![Diagram showing aligned and non-sequential memory access patterns and corresponding memory transactions for different compute capabilities and memory transactions.]
COALESCED MEMORY ACCESS

\[
\begin{pmatrix}
  a_{11} & \cdots & a_{1N} \\
  \vdots & \ddots & \vdots \\
  a_{N1} & \cdots & a_{NN}
\end{pmatrix}
\begin{pmatrix}
  x_1 \\
  \vdots \\
  x_N
\end{pmatrix}
\]

• As an example of the importance of good memory access patterns, consider a matrix-vector multiplication

• A simple way to parallelize the operation is to assign one thread per row that computes the dot product of the row with the vector

• How should we store the matrix into the memory? Let’s consider the two most natural ways, row-wise and column-wise storage

\[
data = \begin{pmatrix}
  a_{11} & a_{12} & \cdots & a_{1N} & a_{21} & a_{22} & \cdots & a_{NN}
\end{pmatrix}
\]

\[
data = \begin{pmatrix}
  a_{11} & a_{21} & \cdots & a_{N1} & a_{12} & a_{22} & \cdots & a_{NN}
\end{pmatrix}
\]
COALESED MEMORY ACCESS

- The thread id gives the row that this thread will operate with.

- The only difference in the kernels is in the highlighted part where the row and column indices have been swapped because of the different data ordering.

- With $N \sim 10000$, the column-wise version is about 13 times faster than the row-wise version!

```c
__global__ void MatrixVectorRow(double* M, double* x, double* y, int N)
{
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    double res = 0;

    if (id < N)
        for(int i=0; i<N; i++)
            res = res + M[id*N + i] * x[i];

    y[id] = res;
}

__global__ void MatrixVectorCol(double* M, double* x, double* y, int N)
{
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    double res = 0;

    if (id < N)
        for(int i=0; i<N; i++)
            res = res + M[i*N + id] * x[i];

    y[id] = res;
}
```
COALESCED MEMORY ACCESS

• Why is it so much better to store the matrix column-wise instead of row-wise?

• In the column-wise case, the threads access the memory contiguously, while in the row-wise case, the memory accesses are scattered and the resulting memory bandwidth is crippled

\[
\begin{pmatrix}
  a_{11} & a_{12} & \cdots & a_{1N} \\
  a_{21} & a_{22} & \cdots & a_{2N} \\
  a_{31} & a_{32} & \cdots & a_{2N} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{N1} & a_{N2} & \cdots & a_{NN}
\end{pmatrix}
\]

1st iteration

2nd iteration

NOT like this:
\[
data = ( a_{11} \ a_{12} \ \cdots \ a_{1N} \ a_{21} \ a_{22} \ \cdots \ a_{NN} )
\]

But like this:
\[
data = ( a_{11} \ a_{21} \ \cdots \ a_{N1} \ a_{12} \ a_{22} \ \cdots \ a_{NN} )
\]
Occupyancy

- Occupancy is the ratio of the number of resident threads to the maximum number of threads, i.e. it’s a measure of how “full” the GPU is.

- Occupancy depends on:
  - The number of registers needed by a thread.
  - The amount of shared memory needed by a block.
  - The block size.

- For example, in the Tesla M2070, the maximum number of blocks per multiprocessor is 8. However, let’s say our kernel is such that one block requires 20 KB of shared memory. Then only 2 blocks can fit into the SM because there’s only 48 KB of shared memory available.

- A simple tool for evaluating the occupancy of your program is the Occupancy Calculator spreadsheet by NVIDIA.

OCCUPANCY

• Let's recall compiling programs with nvcc

```
[sirot1@triton kickstart]$ nvcc -arch sm_20 -o prefix prefix.cu
```

• To see the shared memory and register usage:

```
[sirot1@triton kickstart]$ nvcc -arch sm_20 --ptxas-options=-v -o prefix prefix.cu
```

ptxas info : 0 bytes gmem
ptxas info : Compiling entry function '__Z3SumPf' for 'sm_20'
ptxas info : Function properties for __Z3SumPf
  0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info : Used 9 registers, 4096 bytes smem, 40 bytes cmem[0]

• These numbers can be put to the occupancy calculator
### OCCUPANCY

#### Graphs:
- **Varying Block Size**
- **Varying Shared Memory Usage**

#### Table:
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>2. Enter your resource usage:</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Threads Per Block</td>
<td>256</td>
</tr>
<tr>
<td>10</td>
<td>Registers Per Thread</td>
<td>20</td>
</tr>
<tr>
<td>11</td>
<td>Shared Memory Per Block (bytes)</td>
<td>20000</td>
</tr>
<tr>
<td>12</td>
<td>(Don't edit anything below this line)</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>3. GPU Occupancy Data is displayed here and in the graphs:</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Active Threads per Multiprocessor</td>
<td>512</td>
</tr>
<tr>
<td>15</td>
<td>Active Warps per Multiprocessor</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>Active Thread Blocks per Multiprocessor</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>Occupancy of each Multiprocessor</td>
<td>33%</td>
</tr>
<tr>
<td>20</td>
<td>Physical Limits for GPU Compute Capability:</td>
<td>2.0</td>
</tr>
<tr>
<td>21</td>
<td>Threads per Warp</td>
<td>32</td>
</tr>
<tr>
<td>22</td>
<td>Warps per Multiprocessor</td>
<td>48</td>
</tr>
<tr>
<td>23</td>
<td>Threads per Multiprocessor</td>
<td>1536</td>
</tr>
<tr>
<td>24</td>
<td>Thread Blocks per Multiprocessor</td>
<td>8</td>
</tr>
<tr>
<td>25</td>
<td>Total # of 32-bit registers per Multiprocessor</td>
<td>32768</td>
</tr>
<tr>
<td>26</td>
<td>Register allocation unit size</td>
<td>64</td>
</tr>
<tr>
<td>27</td>
<td>Register allocation granularity</td>
<td>warp</td>
</tr>
<tr>
<td>28</td>
<td>Shared Memory per Multiprocessor (bytes)</td>
<td>49152</td>
</tr>
<tr>
<td>29</td>
<td>Shared Memory Allocation unit size</td>
<td>128</td>
</tr>
<tr>
<td>30</td>
<td>Warp allocation granularity (for register allocation)</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>Allocation Per Thread Block</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>Warps</td>
<td>8</td>
</tr>
<tr>
<td>35</td>
<td>Registers</td>
<td>5120</td>
</tr>
<tr>
<td>36</td>
<td>Shared Memory</td>
<td>20096</td>
</tr>
<tr>
<td>37</td>
<td>These data are used in computing the occupancy data in blue</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>Maximum Thread Blocks Per Multiprocessor</td>
<td>Blocks</td>
</tr>
<tr>
<td>39</td>
<td>Limited by Max Warps / Blocks per Multiprocessor</td>
<td>6</td>
</tr>
<tr>
<td>40</td>
<td>Limited by Registers per Multiprocessor</td>
<td>6</td>
</tr>
<tr>
<td>41</td>
<td>Limited by Shared Memory per Multiprocessor</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>Thread Block Limit Per Multiprocessor highlighted</td>
<td>RED</td>
</tr>
<tr>
<td>43</td>
<td>CUDA Occupancy Calculator</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>Version:</td>
<td>2.1</td>
</tr>
</tbody>
</table>

**Copyright and License**
- Calculator / Help / GPU Data / Copyright & License
OCCUPANCY

• To see you kernels’ resource usage, add the flag `--ptxas-options=-v` to nvcc when compiling and it will list the register and shared memory usage for each kernel.

• Input the numbers along with you block size into the orange box in the occupancy calculator and it will show you the limiting factor in your occupancy.

• If your occupancy is very low (<33%), it might be worthwhile to try to change your kernel to use less registers or shared memory so that more blocks can run concurrently.

• However, it’s not usually important to get 100% occupancy because even at lower occupancies there is enough computation to hide the memory latency.
OPTIMIZING DATA TRANSFER

• A potential bottleneck in a CUDA program is the low bandwidth of the PCIe bus

• This means that sometimes it’s useful to run kernels that have little to no speedup compared to their CPU versions in order to keep the data on the GPU and avoid transfers

• It’s also possible to speed up the cudaMemcpy by using **pinned host memory**
  
  • When a memory transfer is issued, the CUDA driver first allocates a temporary pinned host array, copies the data there and only then from the pinned memory to the GPU
  
  • We can skip the first step by explicitly allocating the pinned memory ourselves
OPTIMIZING DATA TRANSFER

• Pinned memory can be allocated with `cudaMallocHost` (same syntax as `cudaMalloc`)

• You should be careful to not allocate too much pinned memory because it might crash your system

• `cudaMemcpy` may be twice as fast or more with pinned memory, depending on the system
int main() {
    int N = 10000000;
    int numThreadsPerBlock = 256;
    int numBlocks = N / 256 + 1;

    int* A;
    int* B;
    cudaMallocHost((void**)&A, N*sizeof(int));
    cudaMallocHost((void**)&B, N*sizeof(int));

    int* d_A;
    int* d_B;
    cudaMalloc((void**)&d_A, N*sizeof(int));
    cudaMalloc((void**)&d_B, N*sizeof(int));

    for(int i=0; i<N; i++)
        A[i] = i;

    cudaMemcpy(d_A, A, N*sizeof(int), cudaMemcpyHostToDevice);
    Vcopy<<<numBlocks, numThreadsPerBlock>>>(d_A, d_B, N);
    cudaMemcpy(B, d_B, N*sizeof(int), cudaMemcpyDeviceToHost);

    for(int i=0; i<N; i++)
        if(A[i] != B[i])
            std::cout << "Error." << std::endl;
}

• The vector copy example with pinned host memory
STREAMS

- Streams are sequences of operations that are executed on the device in the order that they were issued.
- Operations in different streams don't have to be synchronized, and their execution can be interleaved.
- In the examples thus far, we have implicitly used the default stream, which is a special case in that it blocks the execution of all other streams.
  - However, kernel calls are asynchronous with respect to the CPU.
- By using different streams, it's possible to overlap memory transfers and kernel execution.

```c
cudaStream_t stream1;
cudaStreamCreate(&stream1);

cudaMemcpyAsync(d_A, A, N, cudaMemcpyHostToDevice, stream1);

kernel<<<1, N, 0, stream1>>>(d_A);

cudaStreamDestroy(stream1);
```
OVERLAPPING COMPUTATION AND TRANSFER

- By default, CUDA operations such as kernel calls and memory transfers wait until the previous operation is done before starting the next one.

- However, it is possible to use **asynchronous** versions of these to, for example, overlap transferring memory and running a kernel.

- To overlap, they have to be issued in different, non-default streams.

- The host memory used in the transfer has to be pinned memory.

- The best strategy for overlapping depends on the device but on Fermi devices (which comprise most of the GPU resources currently available), the best way is given in the following example.
```c
int main()
{
    double* A;
    double* d_A;

    int blocksize = 512;
    int N = 16*1024*1024;
    int nstreams = 4;
    int streamsize = N / nstreams;
    int numblocks = streamsize / blocksize;

    cudaEvent_t start1, stop1;
    float time1;

    cudaMallocHost((void**)&A, N*sizeof(double));
    cudaMalloc((void**)&d_A, N*sizeof(double));

    cudaStream_t* streams = new cudaStream_t[nstreams];

    for(int i=0; i<nstreams; i++)
    {
        cudaStreamCreate(&streams[i]);
    }

    cudaEventCreate(&start1);
    cudaEventCreate(&stop1);
    cudaEventRecord(start1,0);

    for(int i=0; i<nstreams; i++)
    {
        cudaMemcpyAsync(d_A+i*streamsize, A+i*streamsize, streamsize*sizeof(double), cudaMemcpyHostToDevice, streams[i]);
        kernel<<<numblocks,blocksize,0,streams[i]>>>(d_A, streamsize);
        cudaMemcpyAsync(A+i*streamsize, d_A+i*streamsize, streamsize*sizeof(double), cudaMemcpyDeviceToHost, streams[i]);
    }

    cudaEventRecord(stop1,0);
    cudaEventSynchronize(stop1);
    cudaEventElapsedTime(&time1, start1, stop1);

    std::cout << "Time: " << time1/1000 << " s" << std::endl;

    for(int i=0; i<nstreams; i++)
    {
        cudaStreamDestroy(streams[i]);
    }
}
```

**Figure 11:** A better version of the vector copy program from Figure 6, suitable for copying large vectors. The vector length is again $N$, and the kernel launch configuration parameters should be chosen in a way that $numBlocks 	imes numThreadsPerBlock$ is equal to or larger than $N$, so that we have enough threads to copy the whole vector. In the kernel, the global ID of the thread is stored into the variable $id$. Then, only threads whose global ID is less than the vector length do the copying. Removing the if statement would result in array overflow, unless $N$ happens to be divisible by $blockDim$, such that we could launch exactly $N$ threads.
```cpp
main()
{
    double A;
    double d_A;
    int blocksize = 512;
    int N = 16 * 1024 * 1024;
    int nstreams = 16;
    int streamsize = N / nstreams;
    int numblocks = streamsize / blocksize;
    cudaEvent_t start1, stop1;
    float time1;
    cudaMallocHost((void **)&A, N * sizeof(double));
    cudaMalloc(d_A, N * sizeof(double));
    cudaStream_t *streams = new cudaStream_t[nstreams];
    for (int i = 0; i < nstreams; i++)
    {
        cudaStreamCreate(&streams[i]);
        cudaEventCreate(&start1);
        cudaEventCreate(&stop1);
        cudaEventRecord(start1, 0);
        for (int i = 0; i < nstreams; i++)
        {
            cudaMemcpyAsync(d_A + i * streamsize, A + i * streamsize,
                             streamsize * sizeof(double), cudaMemcpyHostToDevice,
                             streams[i]);
            kernel<<<numblocks, blocksize, 0, streams[i]>>> (d_A, streamsize);
            cudaMemcpyAsync(A + i * streamsize, d_A + i * streamsize,
                             streamsize * sizeof(double), cudaMemcpyDeviceToHost,
                             streams[i]);
        }
        cudaEventRecord(stop1, 0);
        cudaEventSynchronize(stop1);
        cudaEventElapsedTime(&time1, start1, stop1);
        std::cout << "Time: " << time1 / 1000 << " s" << std::endl;
    }
    cudaEventDestroy(start1);
    cudaEventDestroy(stop1);
    cudaStreamDestroy(streams);
}
```

*Figure 11: A better version of the vector copy program from Figure 6, suitable for copying large vectors. The vector length is again N, and the kernel launch configuration parameters should be chosen in a way that numBlocks times numThreadsPerBlock is equal to or larger than N, so that we have enough threads to copy the whole vector. In the kernel, the global ID of the thread is stored into the variable id. Then, only threads whose global ID is less than the vector length do the copying. Removing the if statement would result in array overflow, unless N happens to be divisible by blockDim, such that we could launch exactly N threads.*
OVERLAPPING COMPUTATION AND TRANSFER

• Sequential version (1 stream)

• 4 streams

• Assuming the kernel takes about the same time as the memory transfer, the execution time is roughly 50%

• More details at for example https://developer.nvidia.com/content/how-overlap-data-transfers-cuda-cc
SUMMARY OF PART II

• Accessing global memory in a way that allows memory transfers to be coalesced is critical for obtaining the best performance.

• As a rule of thumb, always try to arrange your data so that threads of the same warp access data that are close to each other in the memory.

• Managing the use of shared memory and registers is important to achieve high enough occupancy of the multiprocessors and avoid register spilling.
  • Use the occupancy calculator spreadsheet to find out your occupancy.

• Try to minimize transferring data between the host and the device.

• Use pinned host memory to speed up the transfers.

• Streams can be used to overlap memory transfers with kernel execution.
Included in CUDA are many useful libraries, for example:

- **cuFFT** - Fast Fourier transforms
- **CUBLAS** - CUDA-accelerated linear algebra
- **cuRAND** - Random number generation
- **Thrust** - High level interface for many parallel algorithms

With these, you can use the computational power of the GPUs for many basic tasks without having to write any kernels yourself.
CUBLAS

• A CUDA-accelerated version of the standard BLAS library of linear algebra routines

• Support single, double, complex and double complex datatypes

• `#include <cublas.h>`

• Remember to link cublas in the compilation

• CUBLAS documentation: http://docs.nvidia.com/cuda/cublas/index.html
#include <iostream>
#include <cuComplex.h>
#include <cublas.h>

int main()
{
    int N = 10000000;
    cuComplex* A;
    cuComplex* B;
    cudaMallocHost((void**)&A, N*sizeof(cuComplex));
    cudaMallocHost((void**)&B, N*sizeof(cuComplex));

    cuComplex* d_A;
    cuComplex* d_B;
    cudaMalloc((void**)&d_A, N*sizeof(cuComplex));
    cudaMalloc((void**)&d_B, N*sizeof(cuComplex));

    for(int i=0; i<N; i++)
    {
        A[i].x = 0.0;
        A[i].y = i;
    }

    cudaMemcpy(d_A, A, N*sizeof(cuComplex), cudaMemcpyHostToDevice);
    cublasCcopy(N, d_A, 1, d_B, 1);
    cudaMemcpy(B, d_B, N*sizeof(cuComplex), cudaMemcpyDeviceToHost);
}

Figure 16: The prefix sum kernel (following Ref. 20).
THRUST

• Thrust is like a GPU version of the C++ Standard Template Library (STL)

• Quick start guide at http://code.google.com/p/thrust/wiki/QuickStartGuide
CURAND

• A library for CUDA-accelerated random number generation

• You can generate random numbers in bulk from the host code or generate individual numbers inside kernels

• Includes many different generation algorithms and distributions

• More information on the CURAND user guide at http://docs.nvidia.com/cuda/curand/index.html
Hands-on exercise 4 is a code that uses CURAND to approximate pi by generating random points inside a square.

Pi can be approximated from the probability that the point falls inside a quarter circle centered at the lower left corner of the square.

The probability is equal to the ratio of the areas of the quarter circle and the square which is pi/4.

So we generate the points, check the fraction that falls inside the circle, and multiply that by 4 to get an approximation for pi.
HANDS-ON II
PART III
PROFILING
PROFILING

• In CUDA there is a very handy profiler called nvprof that you can use to check the time that is spent in kernels and memory transfers.

```bash
[sirot1@triton kickstart]$ module list
Currently Loaded Modulefiles:
   1) /cuda/5.0
[sirot1@triton kickstart]$ salloc -t 1:00:00 -p gpu --gres=gpu:1 srun --gres=gpu:1 nvprof ./prefix
salloc: Pending job allocation 4298880
salloc: job 4298880 queued and waiting for resources
salloc: job 4298880 has been allocated resources
salloc: Granted job allocation 4298880
======== NVPROF is profiling prefix...
======== Command: prefix
PASSED!
======== Profiling result:
                  Time  Calls  Avg   Min  Max     Name
Time(%)    Time Calls   Avg   Min     Max     Name
72.07   9.33us    1  9.33us 9.33us   9.33us  Sum(float*)
14.58   1.89us    1  1.89us 1.89us   1.89us [CUDA memcpyDtoH]
13.35   1.73us    1  1.73us 1.73us   1.73us [CUDA memcpyHtoD]
salloc: Relinquishing job allocation 4298880
[sirot1@triton kickstart]$  
```

• More details with `--print-gpu-trace` and `--print-api-trace` options to nvprof.
A more advanced tool is the Visual Profiler (nvvp)

The best way to use nvvp is to run it on the same machine with the GPU

On a system like Triton, you have to either

- Log on to the test node gpu001 and run your program through nvvp there
- Run your program normally through the job system with `nvprof --output-profile <filename.out>`
  - This will produce the timeline data that you can import into nvvp

In nvvp, you can see the timeline of what the GPU is doing during the program execution, as well as do a detailed analysis of performance limiting factors, occupancy, memory access patterns etc.
PROFILING

Kernel Performance is Bound By Memory Bandwidth

For device "Tesla M2090" the kernel's compute utilization is significantly lower than its memory utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by the memory system. For this kernel the limiting factor in the memory system is the bandwidth of the Device memory.
LEARN MORE

• CUDA C Programming Guide

• All kinds of resources including video lectures and online courses:
  • https://developer.nvidia.com/cuda-education-training

• NVIDIA Developer Blog
  • https://developer.nvidia.com/blog