# CUDA Advanced Features and CUDA Libraries

Nazım Dugan
nazim.dugan@unibas.ch

# CUDA Advanced Features

# Pinned (Page-locked) memory

• Bandwidth between host memory and device memory is higher if host memory is allocated as page-locked. (see **examples/pinnedMemory.cu**)

• Copies between page-locked host 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 it to or from device memory. (see **examples/zeroCopy.cu**)

**Usage:** Allocate with **cudaMallocHost** function instead of **malloc** or **new**:

```
cudaMallocHost((void**) &data, size * sizeof(double));
```

Free memory using **cudaFreeHost:** `cudaFreeHost(data);`

NVIDIA SDK **bandwithTest** program:

```
Device 0: Tesla C2050
Quick Mode

Host to Device Bandwidth, 1 Device(s), Pinned memory
  Transfer Size (Bytes)        Bandwidth(MB/s)
  33554432                     5853.4

Device to Host Bandwidth, 1 Device(s), Pinned memory
  Transfer Size (Bytes)        Bandwidth(MB/s)
  33554432                     6232.9

Device to Device Bandwidth, 1 Device(s)
  Transfer Size (Bytes)        Bandwidth(MB/s)
  33554432                     84561.2
```

*--memory=pageable*

```
Device 0: Tesla C2050
Quick Mode

Host to Device Bandwidth, 1 Device(s), Paged memory
  Transfer Size (Bytes)        Bandwidth(MB/s)
  33554432                     4234.6

Device to Host Bandwidth, 1 Device(s), Paged memory
  Transfer Size (Bytes)        Bandwidth(MB/s)
  33554432                     3723.0

Device to Device Bandwidth, 1 Device(s)
  Transfer Size (Bytes)        Bandwidth(MB/s)
  33554432                     84584.8
```

# CUDA Streams

Different streams may execute their commands out of order with respect to one another or concurrently.

Mainly used for concurrent kernel execution and overlapping data transfers with kernel execution.

**Usage:**

```
cudaStream_t stream[nStreams];

for (int i=0; i<nStreams; i++)
        cudaStreamCreate(&stream[i]);

cudaMemcpyAsync(data_dev, data, dataSize * sizeof(double), cudaMemcpyHostToDevice, stream[0])
kernel <<< gridSize, blockSize, sharedMemSize, stream[1] >>> (arguments);


for (int i = 0; i < nStreams; ++i)
        cudaStreamDestroy(stream[i]);
```

**cudaMemcpyAsync** function is a non-blocking variant of **cudaMemcpy** in which control is returned immediately to the host thread. It requires a stream to be given as argument and pinned memory should be allocated using **cudaMallocHost** for using this function.

# CUDA Events

- Measure elapsed time for CUDA calls
- Query the status of an asynchronous CUDA call
- Block until CUDA calls prior to the event are completed

**Usage:**

```
// create
cudaEvent_t event1;
cudaEventCreate(&event1);

// record
cudaEventRecord(event1, streamName);

// destroy
cudaEventDestroy(event1);
```

**cudaEventRecord:**
If stream is non-zero, the event is recorded after all preceding operations in the stream have been completed; otherwise, it is recorded after all preceding operations in the CUDA context have been completed.

**Timing:**

```
cudaEventElapsedTime(&floatVariable, event2, event1);
```

**Block stream:**

```
cudaMemcpyAsync(data_dev , data_host,  dataSize, cudaMemcpyHostToDevice, stream1);
cudaEventRecord(event1, stream1);
cudaStreamWaitEvent(stream2, event1, 0);
kernel <<< grid, threads, 0, stream2  >>> (data_dev);
```

**Query:**

```
cudaError_t error = cudaEventQuery(event1);
```

# Overlapping Transfers with Computation and Concurrent Kernel Execution

**Host – Device Computation:** Kernel launches are non-blocking. Therefore, host computation overlaps kernel runs.

```
kernel <<< gridSize, blockSize >>>(arguments);
cpuFunction();
```

**Data transfer – Host Computation:** Asynchronous data transfers can be used to overlap data transfers with host computation.

```
cudaMemcpyAsync(data_dev, data, dataSize * sizeof(double), cudaMemcpyHostToDevice, stream[0])
cpuFunction();
```

**Data transfer – Device Computation:** Asynchronous data transfer and kernel execution can be overlapped if more than one streams are used.

```
cudaMemcpyAsync(data_dev, data, dataSize * sizeof(double), cudaMemcpyHostToDevice, stream[0];
kernel <<< gridSize, blockSize, sharedMemSize, stream[1] >>> (arguments);
```

**Concurrent kernel execution with data transfers:**

```
for (int i=0; i<nStream; i++) {

        int offset = i * dataSize;
        cudaMemcpyAsync(data_dev + offset, data + offset, dataSize * sizeof(double),
                        cudaMemcpyHostToDevice, stream[i];

        kernel <<< gridSize, blockSize, sharedMemSize, stream[i] >>> (arguments);

        cudaMemcpyAsync(data + offset, data_dev + offset, dataSize * sizeof(double),
                        cudaMemcpyDeviceToHost, stream[i];
}
```

# exercise 1

• Given code launches transposeNaive kernel for **M** different **NxN** matrices with a given **nreps** argument. Keep timing data (data transfer to and from device + kernel execution) for **M=96**, **N=2048** and **nreps=20**.

• Modify host memory allocation to use pinned memory and compare total time.

• Modify the code to overlap data transfers and kernel execution. Make the number of streams **nStream** adjustable in such a way that each stream processes (**M/nStream**) matrices. Keep timing data for **nStream=1,2,3,4,8,16,32** to find the optimal value for **nStream**. (Device memory size may avoid large **nStream** values like **32**.)

If your timings are not decreased for **nStream > 1** then your card may not be capable of concurrent data transfers even if it supports overlapping data transfers with kernel execution. For such cards you can try a different strategy as discussed in **NVIDIA CUDA C Programming Guide Section (** *http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#asynchronous-concurrent-execution )*

(GeForce GTX series does not support concurrent data transfers. Tesla series Fermi Cards (Tesla C2050, C2070) supports.)

• Keep timing (1) for kernel execution and (2) for total data transfers, to device and from device, for a single kernel. Then estimate what ratio of data transfers are overlapped with kernel execution in the overlapped version. (Use pinned memory also for timings of (1) and (2).)

# Multiple GPUs

If there are more than one GPU devices in the computation platform then they can be distributed to OpenMP threads or MPI processes.

Get number of GPUs: **cudaGetDeviceCount(&num_gpus)**

Set device for a thread: **cudaSetDevice(gpu_id)**

***example***

```
#pragma omp parallel
{
        // set and check the CUDA device for this CPU thread
        int cpu_thread_id = omp_get_thread_num();
        int gpu_id = -1;
        cudaSetDevice(cpu_thread_id % num_gpus);
        cudaGetDevice(&gpu_id);

        // usual cuda commands
        double *devPtr_dev;
        cudaMalloc((void**) &devPtr, size);

        cudaMemcpy(devPtr, hostPtr + offset, size, cudaMemcpyHostToDevice);

        matmul_shared <<< nBlocks, nThreads >>> (devPtr + ....);

        cudaMemcpy(hostPtr + offset, devPtr, size , cudaMemcpyDeviceToHost);
}
```

**Compile OpenMP :** *nvcc -Xcompiler -fopenmp program.cu*

# exercise 2

Given code does 10 matrix multiplications using multiple GPUs

- Complete the given code to use multiple GPUs with OpenMP parallelization.
- Find optimum work distribution to GPUs

# cudaMallocPitch – cudaMalloc3D

When working with multidimensional arrays if row size is not a multiple of 128 bytes coalescing conditions are not satisfied. In such a case special functions should be used:

**Memory allocation:**
2D : cudaMallocPitch()
3D : cudaMalloc3D()

**Data transfer:**
2D : cudaMemcpy2D()
3D : cudaMemcpy3D()

See **examples/pitchMemory.cu**

***example*** (from NVIDIA C Programming guide)

```
// Host code
int width = 64, height = 64;
float* devPtr;
size_t pitch;
cudaMallocPitch(&devPtr, &pitch, width * sizeof(float), height);
MyKernel<<<100, 512>>>(devPtr, pitch, width, height);

// Device code
__global__ void MyKernel(float* devPtr, size_t pitch, int width, int height)
{
        for (int r = 0; r < height; ++r) {
                float* row = (float*)((char*)devPtr + r * pitch);
                for (int c = 0; c < width; ++c) {
                        float element = row[c];
                }
        }
}
```

# CUDA Libraries

# CUBLAS LIBRARY

GPU enabled blas library : See **CUDA TOOLKIT 4.0 CUBLAS LIBRARY** for details.

**Matrix multiplication example:**

```
#include <cublas.h>
.
.

// allocate host memory and device memory for matrices a,b,c

// initialize cublas
cublasInit();

// copy matrices a and b to device
cublasSetMatrix(n,n,sizeof(double),a_host,lda,a_dev,lda);
cublasSetMatrix(n,n,sizeof(double),b_host,ldb,b_dev,ldb);

//run matrix multiplication
cublasDgemm('N','N',n,n,n,1.0,a_dev,lda,b_dev,ldb,0.0,c_dev,ldc);

// copy result matrix c to host
cublasGetMatrix(n,n,sizeof(double),d_c,ldc,c,ldc);

// shutdown cublas
cublasShutdown();

// free host and device memory
```

**Compile:** *nvcc example.cpp -lcublas* *(-O2 if you have host calculation)*

*icc example.cpp -lcublas -I/usr/local/cuda/include -L/usr/local/cuda/lib64*

# CUBLAS LIBRARY (FORTRAN)

Matrix multiplication example using **thunking** wrapper:

```
program cublas

integer::n,lda,ldb,ldc
real, allocatable::a(:,:),b(:,:),c(:,:)
real*8::t1,t2,tdiff

n = 2560

lda = n
ldb = n
ldc = n

allocate(a(lda,n), b(ldb,n), c(ldc,n))

 call random_number(a)
 call random_number(b)
 call cpu_time(t1)
 call cublas_SGEMM('N','N',n,n,n,1.0,a,lda,b,ldb,0.0,c,ldc)
 call cpu_time(t2)

deallocate(a,b,c)
flops = 2.0 * real(n) * real(n) * real(n)
tdiff = t2-t1
write(*,'(i5,f8.5,i5)') n, tdiff, int(1.0e-9* flops/tdiff)

end
```

Thunking wrapper is easy to use but some slow.

See **examples/cublas_fortran.*** for non-thunking wrapper.

**Compile:** copy **/usr/local/cuda-5.0/src/fortran_thunking.c** to working directory

*icc -I/usr/local/cuda-5.0/include -I/usr/local/cuda-5.0/src -c fortran_thunking.c*

*ifort -L/usr/local/cuda-5.0/lib64 cublas_thunking.f90 fortran_thunking.o -lcublas*

# exercise 3  (15 pts)

- Write your own matrix multiplication program using cublas. Compare its speed with the shared memory matrix multiplication kernel for matrix size of 2560x2560.

  Make the calculation in double precision. You should use **cublasDgemm** function for double precision calculation.

- Now have a single precision version of your program and compare its speed with
- mkl blas cpu calculation with one thread and four threads
  (*export OMP_NUM_THREADS*). You should use **cublasSgemm** function for single precision calculation.

# CUFFT LIBRARY

## Fourier Transform

$$x(t) = \int_{-\infty}^{\infty} \mathcal{F}(f) e^{-2\pi i f t}\, df$$

$$\mathcal{F}(f) = \int_{-\infty}^{\infty} x(t) e^{2\pi i f t}\, dt$$

## Discrete Fourier Transform (DFT)

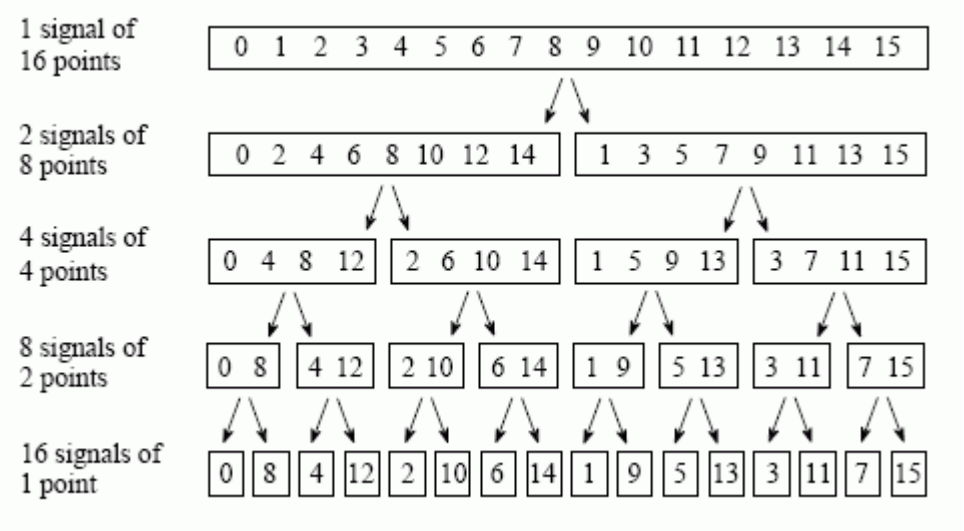
$$A(r) = \sum_{k=0}^{N-1} X(k) \mathbf{W}_N^{rk}$$

$$X(j) = \frac{1}{N} \sum_{k=0}^{N-1} A(k) \mathbf{W}_N^{-jk}$$

$$\mathbf{W}_N = e^{-\frac{2\pi i}{N}}$$

## Fast Fourier Transform (FFT)

Divide and conquer algorithms to compute DFT




**Scaling**

DFT : O(n^2)
FFT : O(n log n)

# CUFFT LIBRARY

## 1D batched forward transform

```
#include <cuda_runtime.h>
#include <cufft.h>

cufftHandle plan;
cufftDoubleComplex *devPtr;
cufftDoubleComplex *data;
cufftResult message;

data = (cufftDoubleComplex*) malloc(dataSize*BATCH * sizeof(cufftDoubleComplex));

/* initialize data here */

cudaMalloc((void**)&devPtr, sizeof(cufftDoubleComplex) * dataSize*BATCH);
cudaMemcpy(devPtr, data, sizeof(cufftDoubleComplex) * dataSize*BATCH, cudaMemcpyHostToDevice);
cufftPlan1d(&plan, dataSize, CUFFT_Z2Z, BATCH);

message = cufftExecZ2Z(plan, devPtr, devPtr, CUFFT_FORWARD);

cudaMemcpy(data, devPtr, sizeof(cufftDoubleComplex) * dataSize*BATCH, cudaMemcpyDeviceToHost);
cufftDestroy(plan);
cudaFree(devPtr);
free(data);
```

**Compile:** *nvcc example.cpp -lcufft*

# CUFFT LIBRARY

## 3D forward transform

```
#include <cuda_runtime.h>
#include <cufft.h>

cufftHandle plan;
cufftDoubleComplex *devPtr;
cufftDoubleComplex *data;
cufftResult message;

data = (cufftDoubleComplex*) malloc(NX*NY*NZ * sizeof(cufftDoubleComplex));

/* initialize data here */

cudaMalloc((void**)&devPtr, sizeof(cufftDoubleComplex) * NX*NY*NZ);
cudaMemcpy(devPtr, data, sizeof(cufftDoubleComplex) * NX*NY*NZ, cudaMemcpyHostToDevice);
cufftPlan3d(&plan, NX,NY,NZ, CUFFT_Z2Z );

message = cufftExecZ2Z(plan, devPtr, devPtr, CUFFT_FORWARD);

cudaMemcpy(data, devPtr, sizeof(cufftDoubleComplex) * NX*NY*NZ, cudaMemcpyDeviceToHost);
cufftDestroy(plan);
cudaFree(devPtr);
free(data);
```

# exercise 4

In this exercise, you will implement 2D FFT using 1D cuFFTs and transpositions for N x N complex data matrix.

**2D FFT requires:**
**Step 1:** N 1D FFTs in the x direction
**Step 2:** N 1D FFTs in the y direction

If data is aligned to be read contiguously in the x direction then, step 2 requires strided access since adjacent elements are separated by N elements. Therefore, straight forward implementation does not have coalesced global memory access for step 2.

Therefore you should do:
**Step 1:** N 1D FFTs in the x direction
**Step 2:** Data transposition
**Step 3:** N 1D FFTs in the x direction
**Step 4:** Data transposition

You can use transposeNoBankConflicts kernel from NVIDIA SDK for the data transpositions. You should make some modifications on this kernel in order to transpose both real and imaginary parts of the data.

Have timing results for your 2D FFT implementation for N=1024, 2048 and compare with 2D cuFFT timing results.

# exercise 5  (10 pts)

2D FFT can be implemented on GPUs without transpositions. In this strategy multiple threads of a block works on multiple columns for the y-direction FFTs.

In this exercise you will use **cufftPlanMany** function to compute 2D transform without transpositions.

Complete the planning stage of forward transform of the given code.

Compare forward transform time with inverse transform (transposed version) time for matrix sizes **512x512** and **2048x2048**

*cufftPlanMany(cufftHandle *plan, int rank, int *n, int *inembed,*
*int istride, int idist, int *onembed, int ostride,*
*int odist, cufftType type, int batch);*

***See:*** *http://docs.nvidia.com/cuda/cufft/index.html#topic_4_1*

# CURAND LIBRARY

## Host API

```
#include <curand.h>
.
.

// define curand generator
curandGenerator_t gen;

// create pseudo-random number generator
curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);

// set seed
curandSetPseudoRandomGeneratorSeed(gen, time(NULL));

// generate double precision random numbers on device
curandGenerateUniformDouble(gen, devData, n);

// transfer results from GPU memory
cudaMemcpy(hostData, devData, n*sizeof(double), cudaMemcpyDeviceToHost);

// use random numbers in host side


// cleanup
curandDestroyGenerator(gen);
```

**Compile:** *nvcc -O2 program.cpp -lcurand*

*icc program.cpp -lcurand -I/usr/local/cuda/include -L/usr/local/cuda/lib64*

# CURAND LIBRARY

## Device API, device code

```
__global__ void setup_kernel(curandState *state, int init) {

        int id = threadIdx.x + blockIdx.x * blockDim.x;

        // Each thread gets different seed, same sequence number
        curand_init(id+init, 0, 0, &state[id]);
}

__global__ void random_kernel(curandState *state, double *result) {

        int gid = threadIdx.x + blockIdx.x * blockDim.x;

        // generate uniform random numbers between 0 and 1
        double random = curand_uniform_double(&state[gid]);

        // use random number
        .
        .

}
```

# CURAND LIBRARY

## Device API, host code

```
#include <curand_kernel.h>
.
.

// allocate space for curand states on device
curandState *devStates;
cudaMalloc((void**)&devStates, nBlocks*nThreads*sizeof(curandState));

// run setup kernel
setup_kernel <<< nBlocks, nThreads >>> (devStates,time(NULL));

// random walk kernel run
random_kernel <<< nBlocks, nThreads >>> (devStates, devResult);

// copy devResult to some host array
```

# exercise 6  (10 pts)

• Use CURAND device API to implement a unidirectional random walk where step size is randomly chosen from interval [0,1].  Set number of steps to 1024x1024 and block size to 1024.

• Initialize CURAND states using the following alternatives and have computation times:

1) Threads get different seeds, same sequence number
2) Threads get same seed, different sequence numbers

• Do the same calculation with host API and compare total times of two versions