IN4049TU: Introduction to Parallel Programming on GPUs

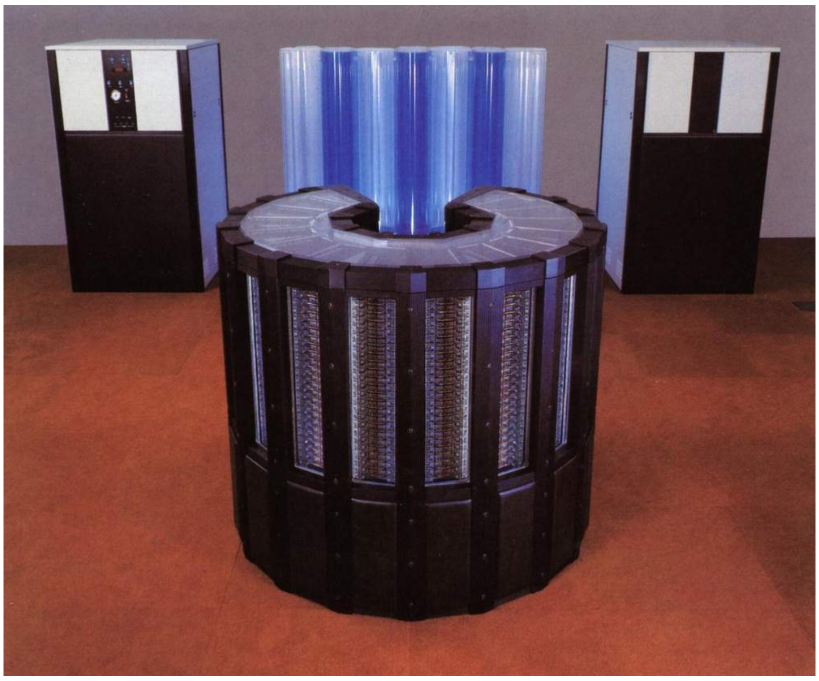
Kees Lemmens,
Email: C.W.J.Lemmens@tudelft.nl,
Delft Institute for Applied Mathematics (DIAM),
Faculty of EEMCS, Delft University of Technology,
The Netherlands.

Historical overview of parallel programming

1960 :	First networks of vacuum-tube computers US Air Force
1985-95:	Super (vector) computers era (Convex, Cray, TM)
1995- :	Beowulf concept : Linux based low budget PC's
2000- :	Linux clusters : Linux based (very) high budget PC's
2004- :	Vector computing on GPU cards designed for gaming
2008- :	High end vector computing on dedicated GPU cards (Nvidia)
2010- :	Multi GPU systems:

- Multicore systems with 4-8 GPUs
- Large GPU clusters with thousands of GPUs

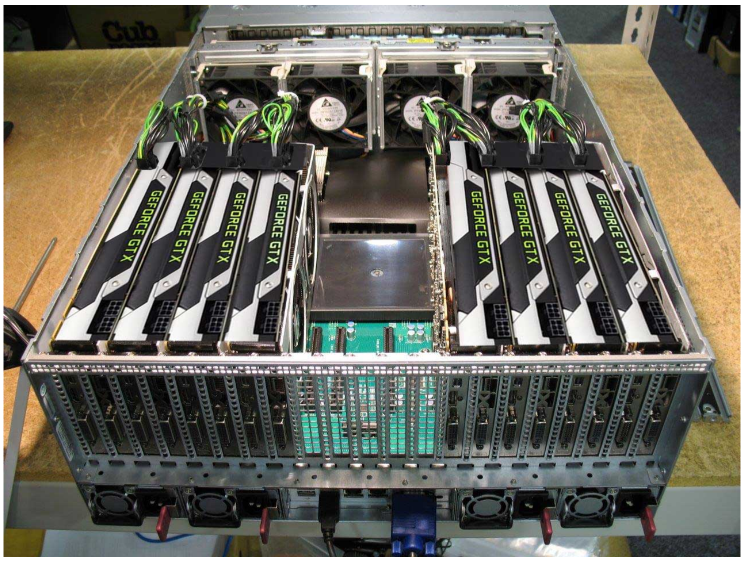
USA: NASA Cray-2 vector Computer, 1.9 Gigaflops, 250 kW (1985)



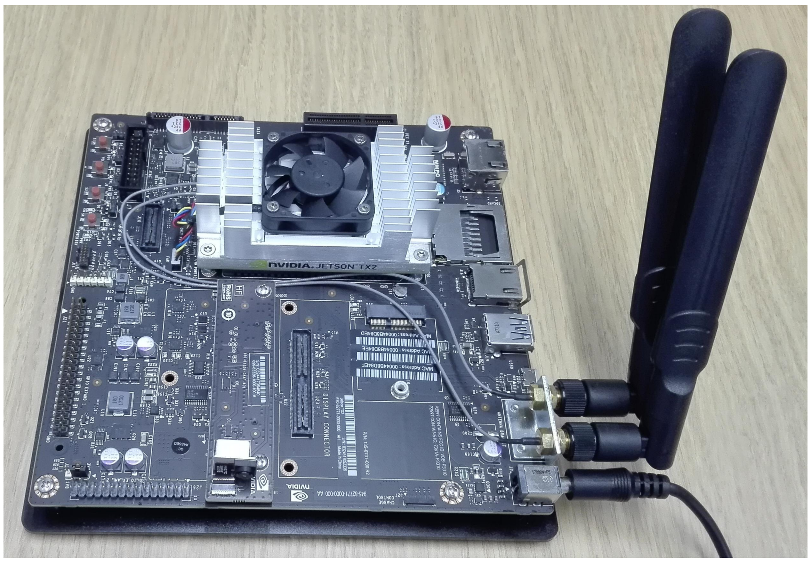
China: Tianhe-2, 34 Petaflops, 24 MWatt, 390 M$, 38.000 Xeon + 44.000 Xeon Phi (2016)

USA: Summit, 150 Petaflops, 13 MWatt, 325 M$, 9200 Power9 + 28.000 Tesla V100 GPU (2019)

USA: Maxwell 8 GPU desktop 20-30 Teraflops (SP), 1.6 kWatt, 15 k$, 8 GPUs GTX980 (2015)



USA San Jose: Jetson Tegra X2 development kit, 30 Watt, 300 $, 1 GPU Pascal on Tegra with 4 core ARM 64 (2017)



Parallel computing on Graphical Processing Units 1

Use graphics (video) cards for scientific computing

Features:

- Based upon standard video cards by Nvidia (or ATI)
- Uses standard computer with either Linux, MSW or MacOS
- Programming model SIMD (Single Instruction, Multiple Data)
- Parallelisation inside card through “threads”
- Dedicated software to access card and start “kernels” (Cuda, OpenCL)

Most popular software is **CUDA** by Nvidia and **OpenCL**, but we’ll mainly talk about **CUDA**, as it is the most commonly used toolkit today.

Parallel computing on Graphics Cards 2

Advantages

- Hardware is (was ?) cheap compared with workstations clusters
- Lower power consumption per Flop than CPU
- Simple GPU already inside many systems without extra investment
- Capable of thousands of parallel threads on a single GPU card
- Very fast for algorithms that can be efficiently parallelised
- Often better scalability than MPI for large problems due to faster communication
- New libraries hiding complexity: **BLAS, FFTW, SPARSE, SOLVER**
- Easy integration with 3D graphics if this runs on the same card (or by using Prime)

Parallel computing on Graphics Cards 3

Disadvantages

- Limited amount of memory available (between 2-16 GByte)
- Memory transfers between host and graphics card add extra overhead
- Often no ECC : error correcting memory (except highend GPUs)
- Fast double precision GPUs still quite expensive
- Slow for algorithms without enough data parallellism
- Debugging code on GPU can be complicated
- Achieving theoretically possible speedup almost impossible
- Using Multi GPUs in a cluster is complex (often with MPI or OpenMP)

What is Cuda ?

means “Compute Unified Device Architecture” and is a software toolkit by Nvidia to ease the use of (Nvidia) graphics cards for scientific programming. It needs on a special video driver as well.

Features :

- Special C compiler to build code both for CPU and GPU (nvcc)
- C Language extensions (codewords) :
  - distinguish CPU and GPU functions
  - access different types of memory on the GPU
  - specify how code should be parallelized on the GPU
- Library routines for memory transfer between host and GPU
- Extra BLAS, Sparse and FFT libraries for easy porting existing code
- Dedicated to Nvidia hardware
- Mainly standard C on the GPU (limited support for C++ and Fortran)

What is OpenCL ?

means “Open Computing Language” and is a C99 like language for parallel kernels in combination with an API to get these kernels loaded and started on dedicated parallel hardware (Apple, Khronos, Nvidia).

Features :

- Only contains a dedicated compiler for building GPU code
- So, no support for mixed (GPU + CPU) source code as in Cuda
- Data transfer between host and GPU done using library functions
- Runs on several types of GPU (AMD, Nvidia) and is easy portable
- However, performance is not necessarily portable across platforms
- OpenCL is an Open Standard, Cuda causes a vendor-lock to Nvidia
- Less mature than Cuda, often no support for latest GPU features
- Starting kernels more complicated than Cuda

When to use GPUs ? Matrix Product comparison

Computation intensive Maxtrix product A x B (single precision).
Time in seconds.

Type	512	1024	2048	4096	8192	16384
Blas/Atlas Xeon 3.6	0.013	0.10	0.78	6.11	48.67	388
CuBlas GPU K600	0.0025	0.016	0.113	0.862	-	-
CuBlas GPU GTX480	0.0016	0.007	0.036	0.222	1.508	-
CuBlas GPU Kepler K20	0.0008	0.0034	0.0155	0.085	0.542	3.94
CuBlas GPU TitanX	0.0007	0.0028	0.0118	0.058	0.327	1.96
Ratio Xeon/TitanX	18x	36x	66x	105x	150x	200x

Conclusions :

- GPUs are much faster than CPUs, especially for larger dimensions.
- GPUs are very limited by their main memory compared to CPUs.
- Gaming cards (GTX480) good price/performance ratio, less memory.

Note : GTX480 and K20 cards in identical hostsystems perform the same.

When to use GPUs ? Matrix Vector Product comparison

Less computation intensive Matrix-Vector product A x b (single precision).
Time in **milli**seconds.

Type (nc=no communication)	512	1024	2048	4096	16384	32768
Blas/Atlas Xeon 3.6	0.1	0.4	2	7	100	500
CuBlas GPU K600	0.6	1	4	20	-	-
CuBlas GPU GTX480	0.4	2	6	30	400	-
CuBlas GPU K20	0.7	1	3	10	200	800
CuBlas GPU TitanX	0.3	0.9	3	10	200	700
CuBlas GPU TitanX (nc)	0.06	0.08	0.1	0.3	4	20
Ratio Xeon/Titan	0.3x	0.5x	0.7x	0.7x	0.5x	0.8x
Ratio Xeon/Titan (nc)	2x	5x	20x	22x	25x	25x

Conclusions :

- GPU performance doesn't improve for larger dimensions.
- Difference in performance for GPUs is only small.
- Blas on CPU faster than on GPU unless no host transfers.

When to use GPUs ? FFT comparison

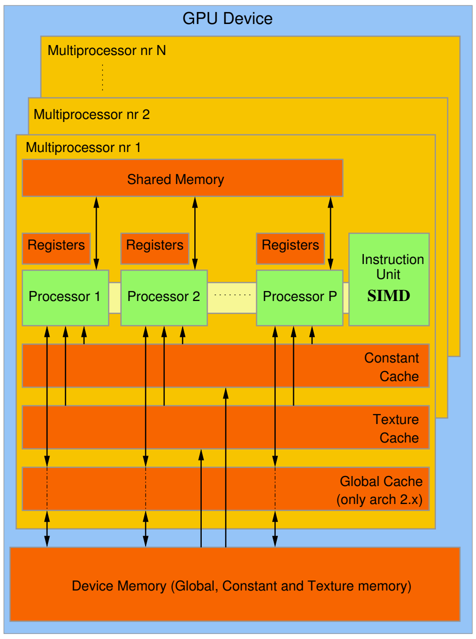
Let's also compare Fast Fourier Transforms on some platforms. For CPUs we used **FFTW** and for GPUs **CuFFT**. Time in **milli**seconds.

Type (nc=no communication)	512	4096	32768	262144	2097152	16777216
CPU Core Duo 1x	0.21	0.35	1.6	16	178	1710
CPU Xeon Oct 1x	0.21	0.34	1.6	15	175	1647
CPU Xeon Oct 50x	0.34	2.60	39.4	477	7040	69790
K600 1x	0.1	0.1	0.6	2.5	15	117
K600 50x (nc)	1.2	1.3	5.2	39.1	243	2002
TitanX 1x	0.2	0.2	0.6	3.9	27	208
TitanX 50x (nc)	1.9	2.3	3.1	5.1	25	176
CPU/GPU 1x TitanX	1x	1.5x	3x	4x	7x	8x
CPU/GPU 50x TitanX (nc)	0.2x	1.1x	13x	94x	280x	397x

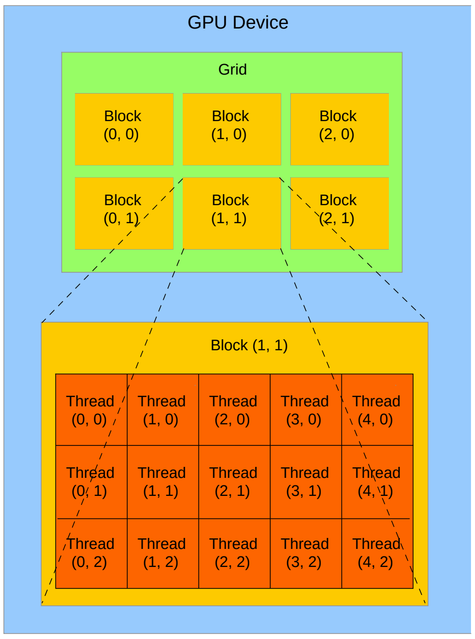
Conclusions :

- Old CPU Core Duo (2010) as fast as an Oct core Xeon (2016) ...
- For small problems cheap GPUs are faster than large GPUs (clock).
- GPU transfer time increases with size, computation time only when full occupation is reached.

Hardware layout of the GPU



Processing layout of a GPU kernel



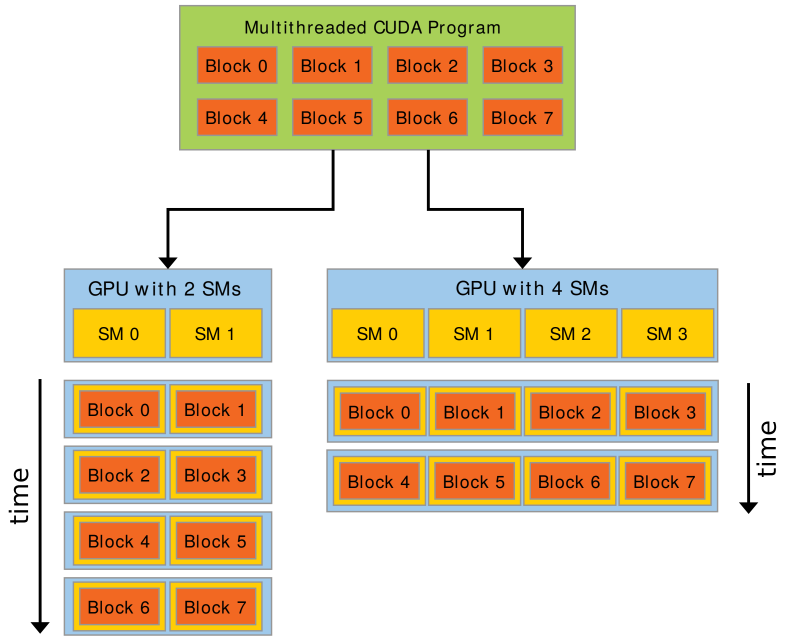
Relation software $\Longleftrightarrow$ hardware

Thread $\Longleftrightarrow$ **Core**

Block $\Longleftrightarrow$ **Multiprocessor**

Grid $\Longleftrightarrow$ **Device**

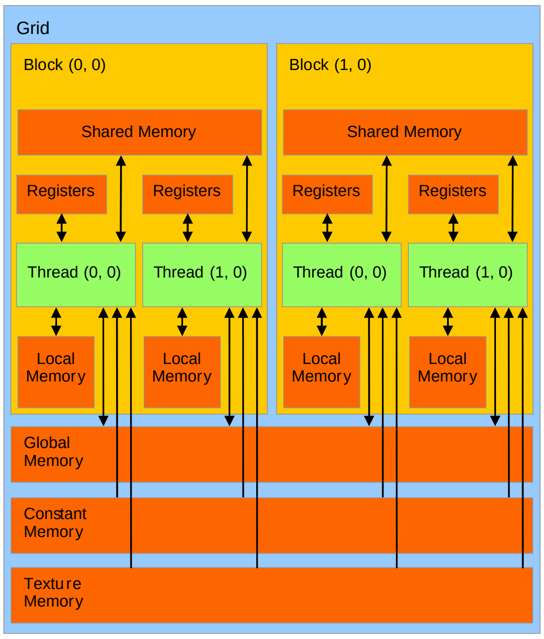
Scalability of a GPU kernel



Some capabilities of an Nvidia GPU

- An Nvidia GPU card can have 1 or more devices (Tesla S1070 = 4, Tesla C2070 = 1, NVS290 = 1, K20 = 1, K80 = 4, Titan X = 1)
- A single device can have several streamprocessors (SM) (NVS290 = 2, K20 = 13, K80 = 2x13, Titan X = 24, P100 = 28, V100=40)
- Each SM has 8 (1.x), 32 (2.0), 48 (2.x), 192 (3.x) or 128 (5/6.x) cores. A SM can run 32 active (semi) concurrent threads (called warps)
- Threads in a SM run concurrently (max. 8-32) or sequentially if more. Maximum resident threads per SM is 768 or 1536 (2.x and up).
- A block is a set of threads running on a single SM. Maximum number of threads per block is 512-1024 (x and y) and 64 (z). The total maximum (x*y*z) is 512 or 1024 (2.x and up)
- Maximum number of blocks is $2^{31} - 1$ for each dimension (for arch 2.x $2^{16} - 1$) independent of SM count. Arch 2.x and up have 3 dimensions (so x, y and z)

Memory organisation of a GPU 1



Memory organisation of a GPU 2

- Device memory consists of global, constant and texture memory
- Global memory accessible by all streaming multiprocessors and is persistent
- Constant and texture memory can only be *read* and have a **local cache** on each SM
- Each SM has shared memory (on-chip) that can be used by all threads in an active block but is non-persistent (kind of cache)
- Each SM has local memory (off-chip) and registers (on-chip) that can only be used by a single thread and which is also non-persistent
- Note that memory access times also depend on the access method ! Coalesced access may give a large performance boost, especially for less computation intensive problems

New features for GPUs architectures

- 1.x or Tesla: Now obsolete.
- 2.x or Fermi: 48 cores per SM, atomic operations. L1 cache (combined with Shared Memory) on each SM and a single L2 cache per device for global memory. Explicit use of shared memory less important.
- 3.x or Kepler: 192 cores per SM, L1 cache only for local memory, Only L2 cache for global memory. Unified Memory Access (no explicit host- device transfers). Explicit use of shared memory again more important.
- 5.x or Maxwell: 128 cores per SM, L1 cache independent from Shared Memory. Dynamic Parallellism: start kernels from a running kernel.
- 6.x or Pascal: 128 cores per SM, Half precision FP (16-bit), NVLink bus to directly link GPUs, bypassing PCI-E bus (5-15 times faster).
- 7.0 or Volta: more half precision FP, Tensor cores (deep learning).
- 7.5 or Turing: mainly graphic and raytracing improvements.
- 8 or Ampere: Tensor core improvements.

Example of parallel work : Amish building a barn

Example of parallel work

If a single Amish can build a barn in 50 days then
50 Amish can build that barn in a single day !

However, in case of ordinary people **and** for GPUs
this is far to optimistic ...

Parallel matrix summation analysis

Suppose we need the sum of 2 (large) matrices A and B :

$$\begin{pmatrix} A(1,1) & \dots & \dots \\ \dots & \dots & \dots \\ \dots & \dots & A(N,N) \end{pmatrix} + \begin{pmatrix} B(1,1) & \dots & \dots \\ \dots & \dots & \dots \\ \dots & \dots & B(N,N) \end{pmatrix} = C$$

How could we parallelize this and what would be the speedup ?

Parallel matrix summation : speedup using strips

Sequential calculation: $N.N$ elements, 1 operation/element :

$$T_{seq} = N^2.t_1 = O(N^2)$$

Parallel calculation for strips : $N.\frac{N}{P}$ elements, 1 op/el :

$$T_{cal} = \frac{N^2}{P}.t_1 = O(\frac{N^2}{P})$$

Parallel communication for strips : scatter A + B + gather C :

$$T_{com} = P.3.\frac{N^2}{P}.t_2 = O(N^2)$$

For $(P > \frac{t_1}{t_2})$: $T_{par} = T_{cal} + T_{com} = O(\frac{N^2}{P}) + O(N^2) \approx O(N^2)$

Speedup : $\frac{T_{seq}}{T_{par}} = \frac{O(N^2)}{O(N^2)} = O(1)$, some for $P < \frac{t_1}{3.t_2}$ (no N dep. !)

Parallel matrix vector product analysis

Suppose we need the product of a (large) matrix A and vector x :

$$\begin{pmatrix} A(1,1) & \dots & \dots \\ \dots & \dots & \dots \\ \dots & \dots & A(N,N) \end{pmatrix} * \begin{pmatrix} x(1) \\ \dots \\ x(N) \end{pmatrix} = b$$

How could we parallelize this and what would be the speedup ?

Parallel matrix vector product analysis : speedup using strips

Sequential calculation: N computations for each element of x of size N :

$$T_{seq} = N^2.t_1 = O(N^2)$$

Parallel calculation for strips : N computations for $\frac{N}{P}$ elements of x :

$$T_{cal} = \frac{N^2}{P}.t_1 = O(\frac{N^2}{P})$$

Parallel communication for strips : scatter $A + x +$ gather b :

$$T_{com} = P.(\frac{N^2}{P} + 2.\frac{N}{P}).t_2 = O(N^2)$$

For $(P >> \frac{t_1}{t_2})$: $T_{par} = T_{cal} + T_{com} = O(\frac{N^2}{P}) + O(N^2) \approx O(N^2)$

Speedup : $\frac{T_{seq}}{T_{par}} = \frac{O(N^2)}{O(N^2)} = O(1)$, some speedup for $P < \frac{t_1}{t_2}$ (no N dep. !)

Parallel matrix multiplication analysis

Suppose we need the product of 2 (large) matrices A and B:

$$\begin{pmatrix} A(1,1) & \dots & \dots \\ \dots & \dots & \dots \\ \dots & \dots & A(N,N) \end{pmatrix} * \begin{pmatrix} B(1,1) & \dots & \dots \\ \dots & \dots & \dots \\ \dots & \dots & B(N,N) \end{pmatrix} = C$$

How could we parallelize this and what would be the speedup ?

Parallel matrix multiplication: speedup using strips

Sequential calculation : $N.N$ elements, N operations/element :

$$T_{seq} = N^3.t_1 = O(N^3)$$

Parallel calculation for strips : $N.\frac{N}{P}$ elements , N op/el :

$$T_{cal} = \frac{N^3}{P}.t_1 = O(\frac{N^3}{P})$$

Parallel communication for strips : scatter A + broadcast B + gather C :

$$T_{com} = P.(2.\frac{N^2}{P} + N^2).t_2 \approx O(P.N^2)$$

For $(N \gg P\frac{t_2}{t_1})$: $T_{par} = T_{cal} + T_{com} = O(\frac{N^3}{P}) + O(P.N^2) \approx O(\frac{N^3}{P})$

Speedup : $\frac{T_{seq}}{T_{par}} = \frac{O(N^3)}{O(\frac{N^3}{P})} = O(P)$, speedup as long as : $P^2 < \frac{N.t_1}{t_2}$

Part 2: Hands on GPU, practical aspects

Control of GPU under CUDA and C: compilers and linkers

To make a Cuda program for the GPU we need some special statements in the code and extra options during compiling and linking.

CUDA provides its own compiler for C: nvcc. It compiles both normal CPU code (for which it simply uses the native C-compiler) and GPU code.

Some things to remember:

- Manually compile using : `% nvcc prg.c -o prg`
- Automatically compile by using e.g. a `Makefile`
- GPU code and CPU code can be in the same source (extension .cu)

Parallel “Hello World1” in Cuda C

A simple Hello World example in C.

```
#include <stdio.h>
#include <cuda.h>

#define NRBLKS   4 // Nr of blocks in a kernel (gridDim)
#define NRTPBK   4 // Nr of threads in a block (blockDim)

__global__ void printOnGPU()
{
   int myid = (blockIdx.x * blockDim.x) + threadIdx.x;
   printf("Hello World, I am thread %d \n",myid);
}

int main(void)
{
   printOnGPU <<< NRBLKS, NRTPBK >>> ();
   cudaDeviceReset(); // Without this line the output is NOT shown !
}
```

Parallel “Hello World1” in Pycuda

A simple Hello World example in Python (but GPU code in C !).

```
import pycuda.autoinit # initializes the device!
from pycuda import compiler, driver

NRBLKS   4 // Nr of blocks in a kernel (gridDim)
NRTPBK   4 // Nr of threads in a block (blockDim)

printOnGPU_code = """
#include <stdio.h>

__global__ void printOnGPU()
{  int myid = (blockIdx.x * blockDim.x) + threadIdx.x;
   printf("Hello World, I am thread %d \n",myid);
}
"""
mod = compiler.SourceModule(printOnGPU_code)
hellogpu = mod.get_function("printOnGPU")

hellogpu(grid=(NRBLKS,1,1), block=(NRTPBK,1,1))
```

Parallel “Hello World2” in Cuda C

```
#include <stdio.h>
#include <cuda.h>
#define NRBLKS   4 // Nr of blocks in a kernel (gridDim)
#define NRTPBK   4 // Nr of threads in a block (blockDim)

__global__ void decodeOnGPU(char *string)
{ int myid = (blockIdx.x * blockDim.x) + threadIdx.x;
  string[myid] ^= (char)011; // 000001001 = 011 in octal
}
int main(void)
{  char *encryption = "Aleef)^f{em)((\11";
   char *string_h, * string_d; // pointers to host and device memory
   int len = strlen(encryption);

   string_h = (char *)malloc(     len);
   cudaMalloc((void **)&string_d, len);
   cudaMemcpy(string_d, encryption, len, cudaMemcpyHostToDevice);
   decodeOnGPU <<< NRBLKS, NRTPBK >>> (string_d);
   cudaMemcpy(string_h, string_d, len, cudaMemcpyDeviceToHost);

   printf("%s\n",string_h);
   cudaFree(string_d);
   free(string_h);
}
```

Unified Memory

A useful addition since architecture 3.0 is Unified Memory. This creates a **single virtual memory space** for all GPU(s) and host(s) memory and **automatically** copies data if accessed from a place where it is not physically located. As such it makes explicit `cudaMemcpy()` calls unnecessary.

Points of attention:

- Key word is `CudaMallocManaged()`
- Can sometimes be slower than manual memory management ...
- Must be compiled with architecture `sm_30` or higher
- Some Intel Memory Management Units may confuse this function, causing wrong answers without any warning!
  Fix: Use boot param `iommu=soft` on Linux. Windows: no idea :)

Example: Discuss and run `hello_unifiedmemory`

Parallel "Hello World2" in Cuda C with unified memory

```
#include <stdio.h>
#include <cuda.h>
#define NRBLKS   4 // Nr of blocks in a kernel (gridDim)
#define NRTPBK   4 // Nr of threads in a block (blockDim)

__global__ void decodeOnGPU(char *string)
{ int myid = (blockIdx.x * blockDim.x) + threadIdx.x;
  string[myid] ^= (char)011; // 000001001 = 011 in octal
}
int main(void)
{  char *encryption = "Aleef)^f{em)((\11";
   char * string; // pointer to unified memory
   int len = strlen(encryption);

   cudaMallocManaged((void **)&string, len);
   strncpy(string, encryption, len);
   decodeOnGPU <<< NRBLKS, NRTPBK >>> (string_d);

   cudaDeviceSynchronize();
   printf("%s\n",string);
   cudaFree(string);
}
```

Generic Cuda program framework

```
#include <stdio.h>
#include <cuda.h>

__global__ void routineOnGPU(float *data, int N)
{
   <GPU code>;
}
int main(void)
{
   float *data_h;  // pointer to host   memory
   float *data_d;  // pointer to device memory
   int size = N * sizeof(float);

   data_h = (float *)malloc(size);
   cudaMalloc((void **) &data_d, size);

   cudaMemcpy(data_d, data_h, size, cudaMemcpyHostToDevice);
   routineOnGPU <<< numberOfBlocks, threadsPerBlock >>> (data_d, N);
   cudaMemcpy(data_h, data_d, size, cudaMemcpyDeviceToHost);

   cudaFree(data_d);
   free(data_h);
}
```

Generic Cuda program framework with Unified Memory

```
#include <stdio.h>
#include <cuda.h>

__global__ void routineOnGPU(float *data, int N)
{
   <GPU code>;
}
int main(void)
{
   float *data;  // pointer to unified host AND device memory
   int size = N * sizeof(float);

   cudaMallocManaged((void **) &data, size);

   routineOnGPU <<< numberOfBlocks, threadsPerBlock >>> (data, N);
   cudaDeviceSynchronize(); // or else no access to "data" from host

   cudaFree(data);
}
```

Another example: Scalar-Vector product (Multiply)

Example: Discussion of a Scalar-Vector product with various amounts of blocks. multiply1 uses simply 1 thread per block, multiply2 uses more.

Test: Let's change the multiplication in `multiply3.cu` into the more time-consuming computation of $sin(\sqrt{nr})$ (changing the $t1/t2$ ratio) and recompile using `make`. Note that we use single precision to keep the results comparable: `sinf(sqrtf(nr))`

Demo: Let's check for both multiply2 and 3 how the ratio between sequential and parallel changes for N=1000 (10^3), N=100000 (10^5) and N=10000000 (10^7) .

Scalar-Vector product (Multiply): K600 vs. Xeon 3.7 GHz

Problem size	N=10^3	N=10^5	N=10^7
Sequential multiply2	1	20	3600
Parallel multiply2	58	361	5500
ratio Par/Seq	76	17	5

Problem size	N=10^3	N=10^5	N=10^7
Sequential multiply3	30	2460	245000
Parallel multiply3	60	373	19000
ratio Par/Seq	2	0.15	0.08

Note : Numbers in microseconds (except of course for the ratios)

Starting a kernel on the GPU

Recall the line from the previous examples :

```
routineOnGPU <<<numberOfBlocks, threadsPerBlock>>>(data_d);
```

Apart from this there are 2 extra and optional arguments :

- Reserve shared memory (often less important if caching is available).
- Associated stream in case more than 1 stream is used.

Up to now we used 2 integers for number of blocks and threads per block. But generally these are of type `dim3` : a 3 dimensional vector with members `dim3.x`, `dim3.y` and `dim3.z`

Max nr of blocks: $2^{16} - 1$ for x and y dims ($2^{31} - 1$ for x, y and z on 3.x).
Max nr of threads: 512 for x and y dims (1024 on 2.x), and 64 for z.

CUDA: get device properties at runtime

To optimise a program for a certain GPU it is often required to obtain runtime information.
This is done with `cudaGetDeviceProperties`:

```
int t,devcount;       // holds number of GPUs
cudaDeviceProp prop;  // holds GPU information

cudaGetDeviceCount(&devcount);

for(t=0;t < devcount; t++)
{ cudaGetDeviceProperties(&prop, t);
  fprintf(stderr,"memory  : %ld Bytes\n",prop.totalGlobalMem);
  fprintf(stderr,"major id: %d \n",prop.major);
  fprintf(stderr,"minor id: %d \n",prop.minor);
}
```

Example: Discussion of `DeviceInfo-GPU` and `DeviceInfo-PY`

Fast Fourier Transforms

Fourier transforms are amongst the most important algorithms invented ! They are used by everyone, everyday, although often without realizing it.
Examples are : Mp3 music, Mpeg video, mobile phones, improving images, MRI scans, oilexploration, Wifi, etc...

According to Fourier every signal (periodic or non-periodic) can be written as a sum of sine and cosine functions, or - easier to use - as complex exponentials. The Fourier or frequency domain simply tells you which frequencies can be found in the original signal and what their relative amplitudes are.

Discrete (Fast) Fourier Transforms or DFTs and FFTs can decompose a (digital) signal into it's individual frequencies, where the number of samples in the signal is exactly equal to the number of frequencies in the decomposition.

Fast Fourier Transforms: formulas

The exact Discrete-time Fourier Transform looks as follows:

$$\boxed{\begin{aligned} \widetilde{f}(\Omega) &= \sum_{n=-\infty}^{\infty} f(n) \cdot e^{-i\Omega n} && \text{Fourier coefficients} \\ f(n) &= \frac{1}{2\pi}\int_0^{2\pi} \widetilde{f}(\Omega) \cdot e^{i\Omega n} d\Omega && \text{Fourier approximation} \end{aligned}}$$

In DFT and FFT summation from $-\infty$ to ∞ is replaced with samples $0 ... (N-1)$ and the continuous integral with a summation over N terms.
If we use the wavenumber k, with the frequency given by $\Omega_k = k \cdot \frac{2\pi}{N}$ we'll find the following equations for both DFT and FFT:

$$\boxed{\begin{aligned} \widetilde{f}(k) &= \frac{1}{N} \sum_{n=0}^{N-1} f(n) \cdot e^{-ik\frac{2\pi}{N}n} && \text{Time to Fourier domain} \\ f(n) &= \sum_{n=0}^{N-1} \widetilde{f}(k) \cdot e^{ik\frac{2\pi}{N}n} && \text{Fourier to Time domain} \end{aligned}}$$

CUDA high level libraries I: the CUFFT library

Cuda contains a straightforward FFT library that works like the popular **FFTW** library, but much faster. To use this you don't need to know much about GPUs as most of the work is done inside the library !

Speedup: close to P for a parallel architecture with P processors.

Example: Discussion of FFt1d on CPU and GPU (framework on next slide)

Exercise: Run the CPU and GPU Fft largetests with vectorsize $2^{18} = 262144$ and just see what the difference in speed is.

Exercise: Now also check N=262143 and N=262145 and compare the results with N=262144.

Note: Both on CPU and GPU the performance is much better for N being a power of 2.

Generic Cuda FFTW program framework

```
#include <cuda.h>
#include <cufft.h>
int main(void)
{
   int dim = 16384;
   cufftComplex *h_t, *h_f;  // time and frequency arrays on host
   cufftComplex *d_t, *d_f;  // time and frequency arrays on device
   int size = dim * sizeof(cufftComplex);
   cufftHandle plan;

   cudaMallocHost(&h_t, size); cudaMallocHost(&h_f, size);
   cudaMalloc    (&d_t, size); cudaMalloc    (&d_f, size);
   cufftPlan1d(&plan, dim, CUFFT_C2C, BATCH);

   cudaMemcpy(d_t, h_t, n *sizeof(cufftComplex), cudaMemcpyHostToDevice);
   cufftExecC2C(plan, d_t, d_f, CUFFT_FORWARD);
   cudaMemcpy(h_f, d_f, n *sizeof(cufftComplex), cudaMemcpyDeviceToHost);

   cufftDestroy(plan);
   cudaFree(d_t);     cudaFree(d_f);
   cudaFreeHost(h_t); cudaFreeHost(h_f);
}
```

Generic Cuda FFTW program framework (Unified Memory)

```
#include <cuda.h>
#include <cufft.h>
int main(void)
{
   int dim = 16384;
   cufftComplex *h_t, *h_f; // time and frequency in managed arrays
   int size = dim * sizeof(cufftComplex);
   cufftHandle plan;

   cudaMallocManaged(&h_t, size); cudaMallocManaged(&h_f, size);
   cufftPlan1d(&plan, dim, CUFFT_C2C, BATCH);

   cufftExecC2C(plan, h_t, h_f, CUFFT_FORWARD);
   cudaDeviceSynchronize();  // must wait until result can be used !

   cufftDestroy(plan);
   cudaFree(h_t); cudaFree(h_f);
}
```

CUDA high level libraries II: the CUBLAS library

Cuda also contains a BLAS library that works much like the popular **BLAS** library, but a great deal faster, especially for level 3 routines. Again most of the gory GPU stuff is done inside the library.

Example: Discussion of `MatrixProd-Blas-GPU` (framework on next slide)

cublasSgemm

Exercise : Run the CPU (Gsl version) and GPU programs for NxN matrices with N=1024, N=4096, N=2048, N=2047 and N=2049 and compare the results.

Note: The difference between 2047 and 2048 can be almost a factor 4 on older GPU architectures ! However, on 2.x and up and on CPU there should hardly be any difference.

Generic Cuda Blas framework Sgemm

```
#include <stdio.h>
#include <cuda.h>
#include <cublas_v2.h>
int main(void)
{
   float *data_hA, *data_hB, *data_hC;  // pointers to host   memory
   float *data_dA, *data_dB, *data_dC;  // pointers to device memory
   cublasHandle_t handle;
   int size = dim * dim * sizeof(float);

   cublasCreate(&handle);
   data_hA = (float *)malloc(size);      // 3 times for A, B and C
   cudaMalloc((void **) &data_dA, size); // 3 times for A, B and C

   cublasSetVector(dim*dim, sizeof(float), data_hA, 1, data_dA, 1); // 2x : A and B
   status = cublasSgemm(handle, ... data_dA, dim, data_dB, dim, ..., data_dC, dim);
   cublasGetVector(dim*dim, sizeof(float), data_dC, 1, data_hC, 1);

   cublasDestroy(handle);
   cudaFree(data_dA); // 3 times for A, B and C
   free(data_hA);     // 3 times for A, B and C
}
```

Generic Cuda Blas framework Sgemv

```
#include <cuda.h>
#include <cublas_v2.h>
int main(void)
{
   float *h_A, *h_x, *h_b;  // matrix A and vectors b,x in host memory
   float *d_A, *d_x, *d_b;  // matrix A and vectors b,x in device memory
   float alpha = 1, beta = 0;
   cublasHandle_t handle;

   cublasCreate(&handle);
   cudaMallocHost(&h_A, dim * dim * sizeof(float));
   cudaMallocHost(&h_x, dim *       sizeof(float)); // and once more for b
   cudaMalloc    (&d_A, dim * dim * sizeof(float));
   cudaMalloc    (&d_x, dim *       sizeof(float)); // and once more for b

   cublasSetVector(dim*dim, sizeof(float), h_A, 1, d_A, 1);
   cublasSetVector(dim    , sizeof(float), h_x, 1, d_x, 1);
   status = cublasSgemv(handle, ... , &alpha, d_A, dim, d_x, 1, &beta, d_b, 1);
   cublasGetVector(dim    , sizeof(float), d_b, 1, h_b, 1);

   cublasDestroy(handle);
   cudaFree    (d_A); cudaFree    (d_x); cudaFree    (d_b);
   cudaFreeHost(h_A); cudaFreeHost(h_x); cudaFreeHost(h_b);
}
```

Generic Cuda Blas framework Sgemv (unified memory)

```
#include <cuda.h>
#include <cublas_v2.h>
int main(void)
{
   float *h_A, *h_x, *h_b;  // matrix A and vectors b,x in managed memory
   float alpha = 1, beta = 0;
   cublasHandle_t handle;

   cublasCreate(&handle);
   cudaMallocManaged(&h_A, dim * dim * sizeof(float));
   cudaMallocManaged(&h_x, dim *       sizeof(float)); // and once more for b

   status = cublasSgemv(handle, ... , &alpha, h_A, dim, h_x, 1, &beta, h_b, 1);
   cudaDeviceSynchronize();  // must wait until result can be used !

   cublasDestroy(handle);
   cudaFree(h_A); cudaFree(h_x); cudaFree(h_b);
}
```

Debugging of memory leak errors using cuda-memcheck

cuda-memcheck is a memory leak checker for CUDA and works much the same as the Unix tool valgrind.

Notes

- Start it with : `% cuda-memcheck <PRG>`
- Cuda-memcheck always stops after the first memory error. You can force it to continue for the rest of the program using :
  `% cuda-memcheck --destroy-on-device-error kernel <PRG>` (5.x and up)

Example: Run the Memcheck-GPU example using memcheck and explain what is wrong in the code there.

Profiling Cuda code

Since CUDA version 5.x there is also a nice commandline profiler nvprof.

How to use: `nvprof myprogram <arguments>`
Kernel details: `nvprof --print-gpu-trace myprogram <arguments>`
Output for nvvp: `nvprof -o myprof.nvvp myprogram <arguments>`

There is also a GUI profiler named nvvp since Cuda 4.1 with more options but nvprof is really fine for basic profiling.

Example: Small demo for `matrixprod-simple` with 2 kernels in nvprof.

Example: Also show `parbody-psm1` (1024grid.par) with the nvvp GUI.

Exercise: Try to run nvprof and possibly nvvp by yourself.

Note: for nvvp set the executable name in File $->$ New Session $->$ File and add optional arguments before starting an analysis.

Some nice examples

Ok, so far the hard part of this Cuda course !

Let's try a few nice (graphic) examples, such as:

- fluidsGL
- smokeParticles
- randomFog
- particles
- Mandelbrot
- bandwidthTest

THE END

Thanks for your attention !