## 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 ? 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.

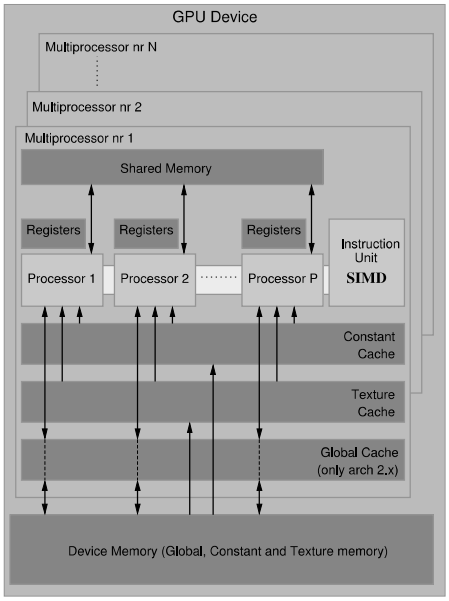
| Type (nc=no communication) | 512 | 4096 | 32768 | 262144 | 2097152 | 16777216 |
|---|---|---|---|---|---|---|
| CPU Core Duo 1x | 0.21 | 0.35 | 1.6 | 16 | 178 | 1710 |
| CPU Xeon Oct 1x | 0.21 | 0.34 | 1.6 | 15 | 175 | 1647 |
| CPU Xeon Oct 50x | 0.34 | 2.60 | 39.4 | 477 | 7040 | 69790 |
| K600 1x | 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 :

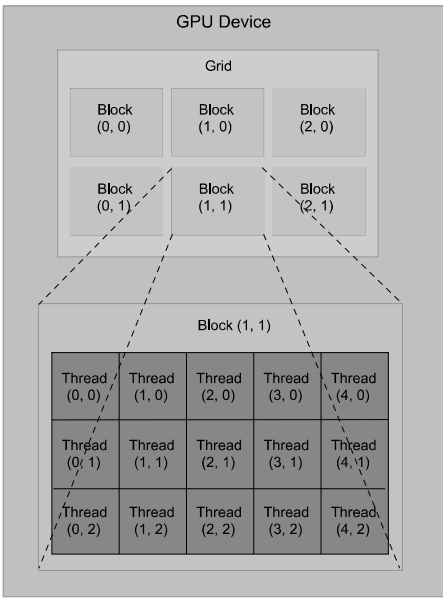
- Old CPU Core Duo (2010) as fast as an Oct core Xeon (2016) ...
- For small problems cheap GPUs are faster than large GPUs (clock).
- GPU transfer time increases with size, computation time only when full occupation is reached.



## Hardware layout of the GPU



## Processing layout of a GPU kernel





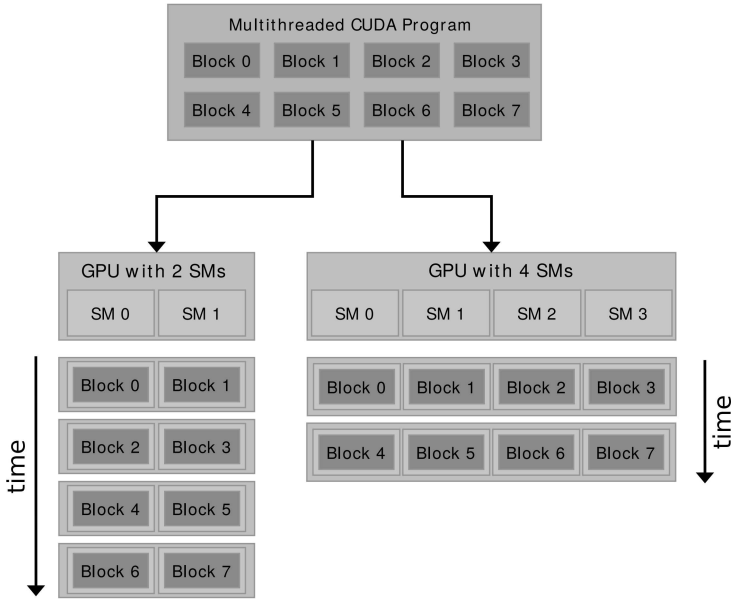

## Relation software $\Longleftrightarrow$ hardware

**Thread** $\Longleftrightarrow$ **Core**

**Block** $\Longleftrightarrow$ **Multiprocessor**

**Grid** $\Longleftrightarrow$ **Device**

## Scalability of a GPU kernel

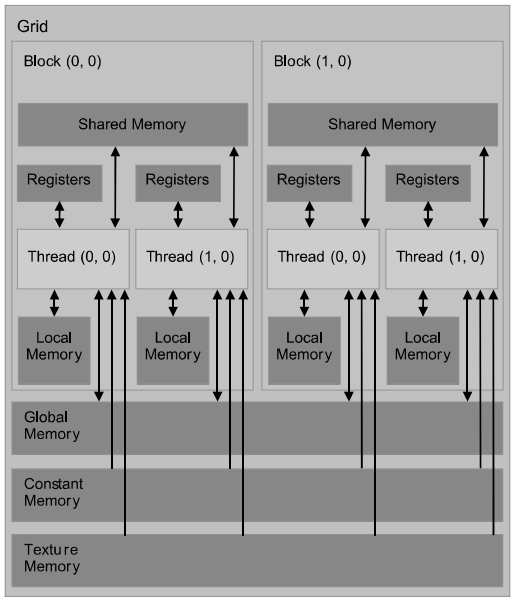





## Some capabilities of an Nvidia GPU

- 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

## 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 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