Programming of Graphics Cards

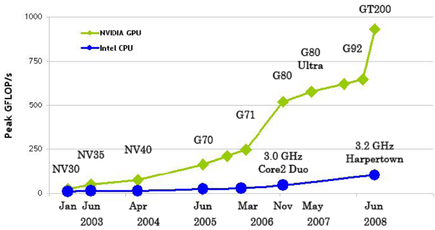
Stefan Lang

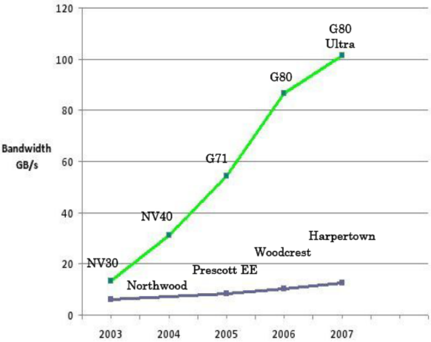
Interdisciplinary Center for Scientific Computing (IWR)
University of Heidelberg
INF 368, Room 532
D-69120 Heidelberg
phone: 06221/54-8264
email: Stefan.Lang@iwr.uni-heidelberg.de

WS 14/15

Motivation

- Development of graphics processors (GPU) is dramatical:





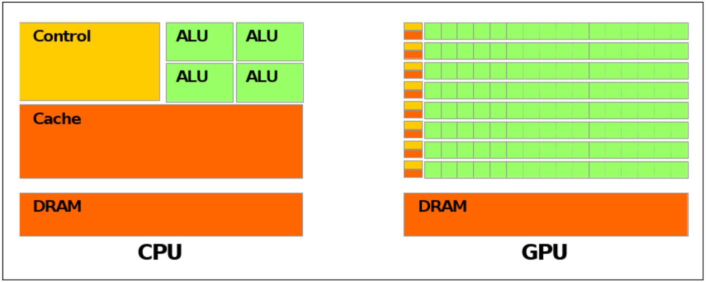
- GPUs are highly parallel processors!
- **GPGPU computing**: Use GPUs for parallel computation.

GPU - CPU Comparison

	Intel QX 9770	NVIDIA 9800 GTX
Since	Q1/2008	Q1/2008
Cores	4	16×8
Transistors	820 Mio	754 Mio
Clock	3200 MHz	1688 MHz
Cache	4×6 MB	16×16 KB
Peak	102 GFlop/s	648 GFlop/s
Bandwith	12.8 GB/s	70.4 GB/s
Price	1200 $	150 $

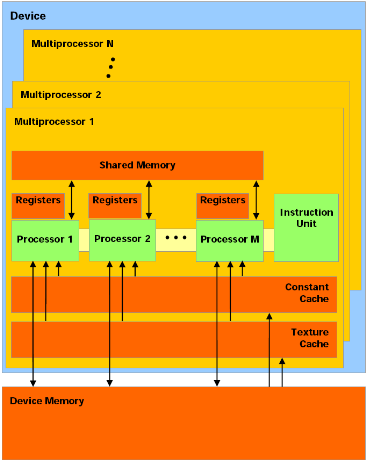
Last model GTX 280 has 30×8 cores and a peak performance of 1 TFLOPs.

Chip Architecture: CPU vs. GPU



GPU tremendously more transistors for data processing, therefore fewer transistors for cache

Hardware on Sight

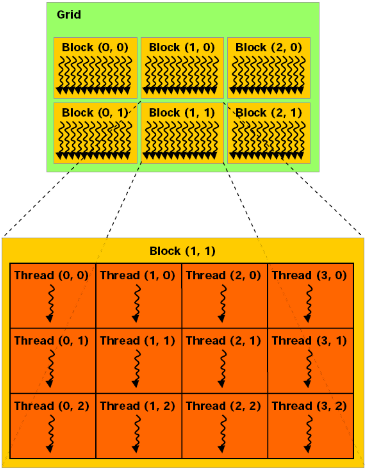


- A multiprocessor (MP) consists of $M = 8$ “processors”.
- MP has an instruction unit and 8 ALUs. Threads, that execute different instructions, are serialised!
- 8192 registers per MP, are divided onto threads at compile time.
- 16 KB shared memory per MP, organised in 16 banks.
- Up to 4 GB global memory, latency 600 clock cycles, bandwidth up to 160 GB/s .
- Constant- and texture memory is cached and is read-only.
- Graphics cards deliver high performance for arithmetics with single precision, double precision lower performance.
- Arithmetics is not (completely) IEEE conforming.

CUDA

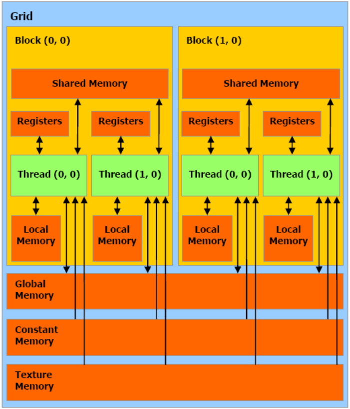
- Stands for **Compute Unified Device Architecture**
- Scalable hardware model with e.g. 4×8 processors in a notebook and 30×8 processors on a high-end card.
- C/C++ programming environment with language extensions. Special compiler `nvcc`.
- The code, executable on the GPU, can only be written in C.
- Runtime environment and different application libraries (BLAS, FFT).
- Extensive set of examples.
- Coprocessor architecture:
  - Some code parts run on the CPU, that then initiates code on the GPU.
  - Data has to be explicitly copied between CPU and GPU memory (no direct access).

Programming Model on Sight



- Parallel threads cooperate with shared variables.
- Threads are grouped in blocks of a “choosable” size.
- Blocks can be 1-, 2- or 3-dimensional.
- Blocks are organized in a grid with variable size.
- Grids can be 1- or 2-dimensional.
- # threads is typically larger than # cores (“hyperthreading”).
- Block size is determined by HW/Problem, grid size is determined by problem size.
- No overhead through context switch.

Memory Hierarchy and Access of Instances



Memory hierarchy with specific access of individual instances (thread, block and grid)

- Per thread
  - Register
  - Local memory (uncached)
- Per block
  - Shared memory
- Per grid
  - Global memory (uncached)
  - Constant memory (read-only, cached)
  - Texture memory (read-only,cached)

Example of a Kernel

```
__global__ void scale_kernel (float *x, float a)
{
  int index = blockIdx.x*blockDim.x + threadIdx.x;
  x[index] *= a;
}
```

- `__global__` function type qualifies this function for execution on the device and can only be called from host (“kernel”).
- Built-in variable `threadIdx` contains position of threads within the block.
- Built-in variable `blockIdx` stores position of block within the grid.
- Built-in variable `blockDim` provides the size of the blocks.
- Built-in variable `gridDim` contains dimension of the grid
- In the example above each thread is responsible to scale an element of the vector.
- The total count of threads has to be adapted to the size of the vector.

Execution and Performance Aspects

- Divergence: Full performance can only be achieved if all threads of a warp execute an identical instruction.
- Threads are scheduled in *warps* of 32 threads.
- Hyperthreading: A MP should execute more than 8 threads at a time (recommended block size is 64) to hide the latency time.
- Shared memory access uses 2 clock cycles.
- Fastest instructions are 4 cycles (e.g. single precision multiply-add).
- Access of shared memory is only fast if each thread accesses a different bank, otherwise the bank access is serialized.
- Access to global memory can be accelerated by collection of the access to aligned memory locations. Necessitates special data types, e.g. `float4`.

Synchronisation / Branching

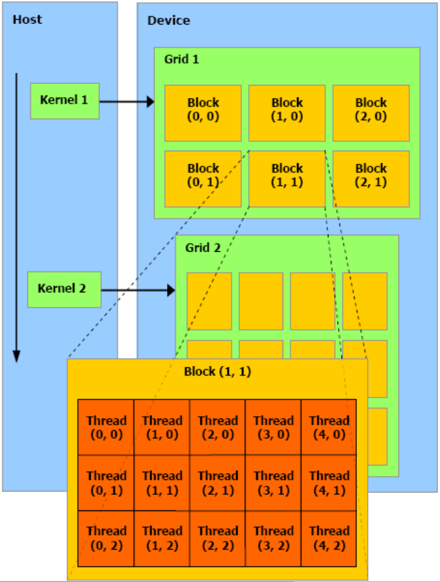
Synchronisation

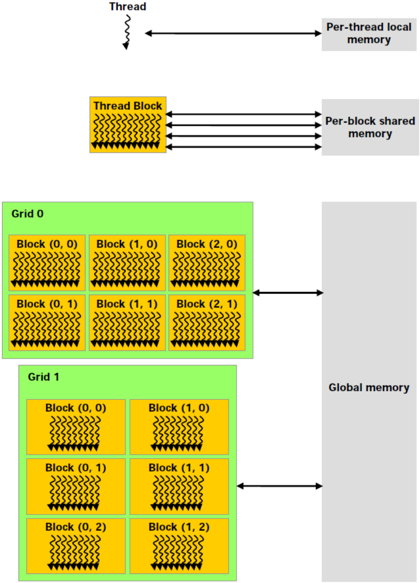
- Synchronisation with barrier on block level.
- No synchronisation mechanisms between blocks.
- But: Kernel calls are cheap, can be used for synchronisation between blocks.
- Atomic operations (not all models from compute capability 1.1).

Branching

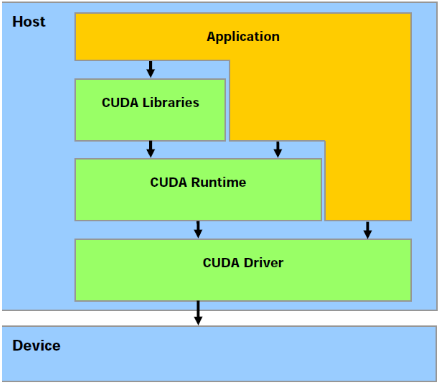
- Each stream processor has its own program counter and can branch individual.
- But: branch divergence within a warps (32 threads) is expensive, deviating threads are executed serially.
- No recursion

Execution Model





CUDA API



- Extensions to standard C/C++
- Runtime environment: Common, components
- Software Development Kit (CUDA SDK) with many examples
- CUFFT and CUBLAS libraries
- Support for Windows, Linux and Mac OS X

CUDA Language Extensions

- Function type delimiter
  - `__device__` on device, callable from device.
  - `__global__` on device, callable from host.
  - `__host__` on host, callable from host (default).
- Variable type delimiter
  - `__device__` in global memory, validity for app.
  - `__constant__` in constant memory, validity for app.
  - `__shared__` in shared memory, validity for block.
- Directive for kernel call (see below).
- Built-in variables `__gridDim__`, `__blockIdx__`, `__blockDim__`, `__threadIdx__`, `__warpSize__` .

CUDA Execution Configuration

- Kernel instantiation:
  kernelfunc «<Dg, Db, Ns»> (arguments)
- dim3 Dg: size of the grid
- Dg.x * Dg.y = number of blocks
- dim3 Db: size of each block
- Db.x * Db.y * Db.z = Number of threads per block
- Ns: byte count of dynamically allocated shared memory per block

Hello CUDA I

```
// scalar product using CUDA
// compile with: nvcc hello.cu -o hello

// includes, system
#include<stdlib.h>
#include<stdio.h>

// kernel for the scale function to be executed on device
__global__ void scale_kernel (float *x, float a)
{
  int index = blockIdx.x*blockDim.x + threadIdx.x;
  x[index] *= a;
}

// wrapper executed on host that calls scale on device
// n must be a multiple of 32 !
void scale (int n, float *x, float a)
{
  // copy x to global memory on the device
  float *xd;
  cudaMalloc( (void**) &xd, n*sizeof(float) ); // allocate memory on device
  cudaMemcpy(xd,x,n*sizeof(float),cudaMemcpyHostToDevice); // copy x to device

  // determine block and grid size
  dim3 dimBlock(32);   // use BLOCKSIZE threads in one block
  dim3 dimGrid(n/32);  // n must be a multiple of BLOCKSIZE!

  // call function on the device
  scale_kernel<<<dimGrid,dimBlock>>>(xd,a);

  // wait for device to finish
  cudaThreadSynchronize();

  // read result
  cudaMemcpy(x,xd,n*sizeof(float),cudaMemcpyDeviceToHost);

```

Hello CUDA II

```
  // free memory on device
  cudaFree(xd);
}

int main( int argc, char** argv)
{
  const int N=1024;
  float sum=0.0;
  float x[N];
  for (int i=0; i<N; i++) x[i] = 1.0*i;
  scale(N,x,3.14);
  for (int i=0; i<N; i++) sum += (x[i]-3.14*i)*(x[i]-3.14*i);
  printf("%g\n",sum);
  return 0;
}
```

Scalarproduct I

```
// scalar product using CUDA
// compile with: nvcc scalarproduct.cu -o scalarproduct -arch sm_11

// includes, system
#include<stdlib.h>
#include<stdio.h>
#include<math.h>
#include<sm_11_atomic_functions.h>

#define PROBLEMSIZE 1024
#define BLOCKSIZE 32

// integer in global device memory
__device__ int lock=0;

// kernel for the scalar product to be executed on device
__global__ void scalar_product_kernel (float *x, float *y, float *s)
{
  extern __shared__ float ss[]; // memory allocated per block in kernel launch
  int block = blockIdx.x;
  int tid = threadIdx.x;
  int index = block*BLOCKSIZE+tid;

  // one thread computes one index
  ss[tid] = x[index]*y[index];
  __syncthreads();

  // reduction for all threads in this block
  for (unsigned int d=1; d<BLOCKSIZE; d*=2)
    {
      if (tid%(2*d)==0) {
        ss[tid] += ss[tid+d];
      }
      __syncthreads();
  }
```

Scalarproduct II

```
    // combine results of all blocks
    if (tid==0)
      {
        while (atomicExch(&lock,1)==1) ;
        *s += ss[0];
        atomicExch(&lock,0);
      }
  }

  // wrapper executed on host that uses scalar product on device
  float scalar_product (int n, float *x, float *y)
  {
    int size = n*sizeof(float);

    // allocate x in global memory on the device
    float *xd;
    cudaMalloc( (void**) &xd, size ); // allocate memory on device
    cudaMemcpy(xd,x,size,cudaMemcpyHostToDevice); // copy x to device
    if( cudaGetLastError() != cudaSuccess)
      {
        fprintf(stderr,"error␣in␣memcpy\n");
        exit(-1);
      }

    // allocate y in global memory on the device
    float *yd;
    cudaMalloc( (void**) &yd, size ); // allocate memory on device
    cudaMemcpy(yd,y,size,cudaMemcpyHostToDevice); // copy y to device
    if( cudaGetLastError() != cudaSuccess)
      {
        fprintf(stderr,"error␣in␣memcpy\n");
        exit(-1);
      }

    // allocate s (the result) in global memory on the device
    float *sd;
    cudaMalloc( (void**) &sd, sizeof(float) ); // allocate memory on device
```

Scalarproduct III

```
float s=0.0f;
cudaMemcpy(sd,&s,sizeof(float),cudaMemcpyHostToDevice); // initialize sum on device
if( cudaGetLastError() != cudaSuccess)
  {
    fprintf(stderr,"error␣in␣memcpy\n");
    exit(-1);
  }

// determine block and grid size
dim3 dimBlock(BLOCKSIZE);   // use BLOCKSIZE threads in one block
dim3 dimGrid(n/BLOCKSIZE);  // n is a multiple of BLOCKSIZE

// call function on the device
scalar_product_kernel<<<dimGrid,dimBlock,BLOCKSIZE*sizeof(float)>>>(xd,yd,sd);

// wait for device to finish
cudaThreadSynchronize();
if( cudaGetLastError() != cudaSuccess)
  {
    fprintf(stderr,"error␣in␣kernel␣execution\n");
    exit(-1);
  }

// read result
cudaMemcpy(&s,sd,sizeof(float),cudaMemcpyDeviceToHost);
if( cudaGetLastError() != cudaSuccess)
  {
    fprintf(stderr,"error␣in␣memcpy\n");
    exit(-1);
  }

// free memory on device
cudaFree(xd);
cudaFree(yd);
cudaFree(sd);

// return result
```

Scalarproduct IV

```
  return s;
}

int main( int argc, char** argv)
{
  float x[PROBLEMSIZE], y[PROBLEMSIZE];
  float s;
  for (int i=0; i<PROBLEMSIZE; i++) x[i] = y[i] = sqrt(2.0f);
  s = scalar_product(PROBLEMSIZE,x,y);
  printf("result of scalar product is %f\n",s);
  return 0;
}
```

Remark: This is not the most efficient version. See the CUDA tutorial for a version that uses the full memory bandwidth.