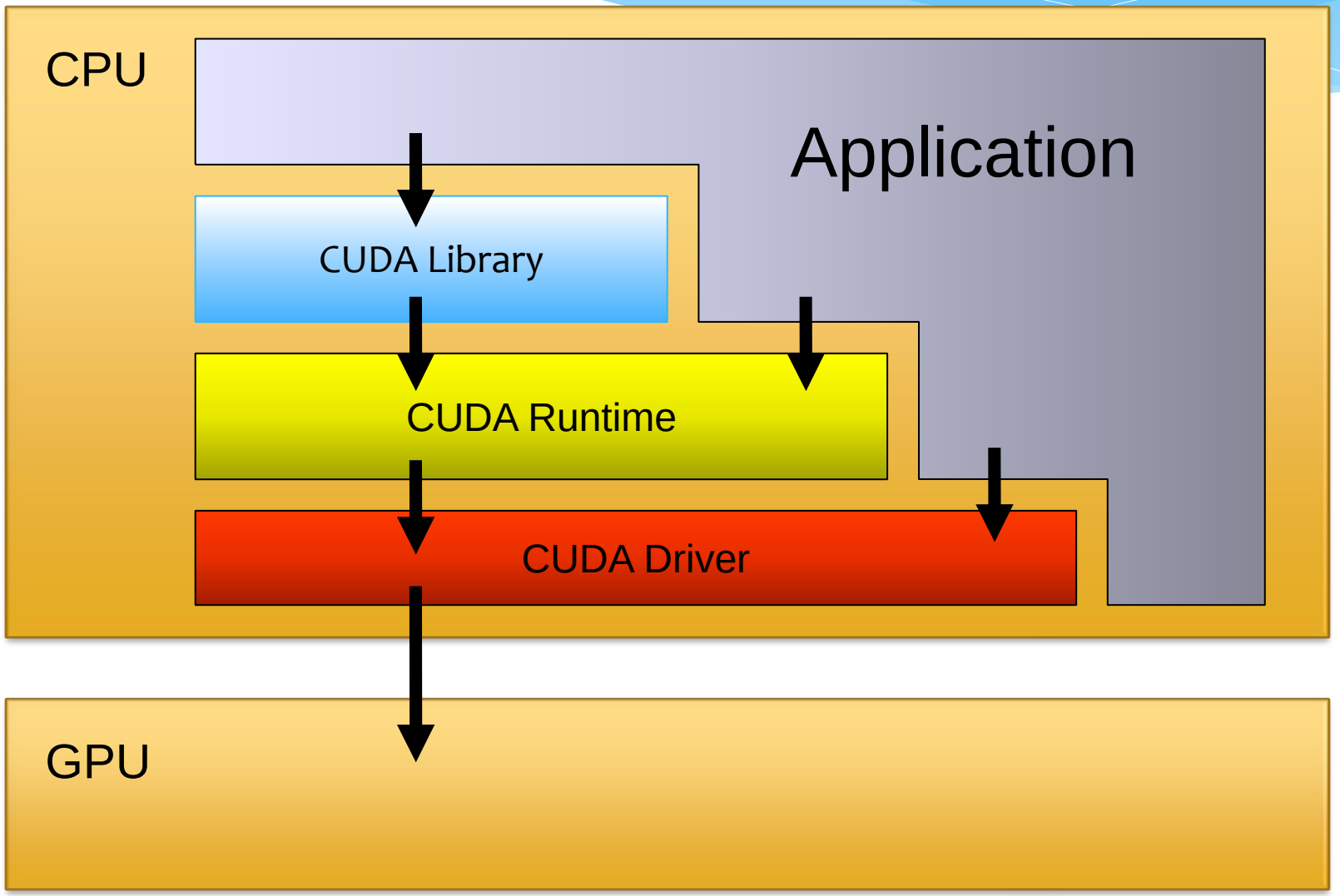
Master Educational Program
“Information technology in applications”

Mathematical computations with GPUs

CUDA

Alexey A. Romanenko
arom@ccfit.nsu.ru
Novosibirsk State University

CUDA - Compute Unified Device Architecture



CUDA Roadmap

2014 – 7 years!

* CUDA 1.0 – 2007
* CUDA 2.0 – 2008
* CUDA 3.0 – 2009
* CUDA 4.0 – 2011
* CUDA 5.0 – 2012
* CUDA 5.5 – 2013
* CUDA 6.0 - 2014

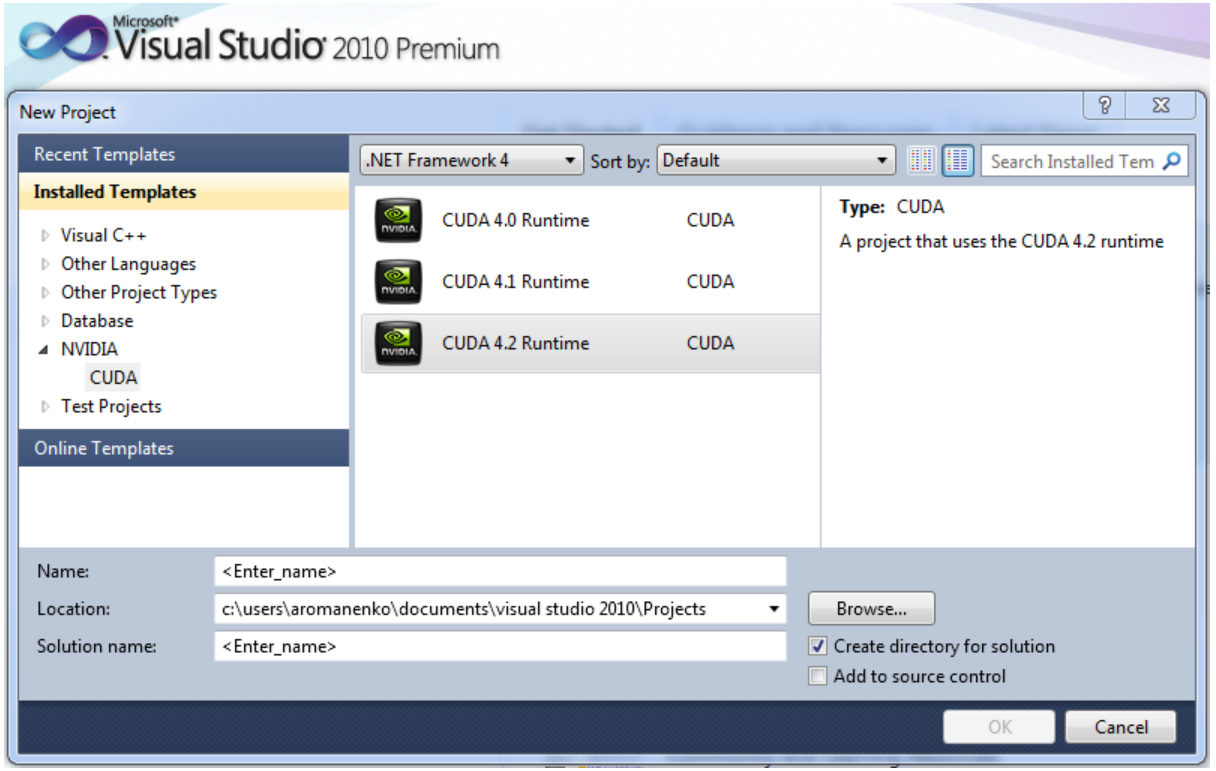
CUDA, components

* Driver (must be installed)
  * /lib/modules/...
* Toolkit (compiler, libraries CUBALS, CUFFT…)
  * /usr/local/cuda
* SDK (examples)
  * /usr/local/cudasdk

Compiling of CUDA program

* Compiler — nvcc
* Source codes- *.cu or *.cuh
* It's recommended to use **make** utility
  * Use Makefile from SDK as an example
  * Modify paths, file names, etc.
* Check **nvcc** options
  * nvcc --help

NSight 2.0+



For CUDA 3.2 + NSight 1.5
http://blog.cuvilib.com/2011/02/24/how-to-run-cuda-in-visual-studio-2010/

* Building process
  * NVCC separate device and host code
  * Device code -> PTX assebler or/and CUBIN
  * Host code: <<<… >>> -> driver API, etc.
* Computer compatibility options and future compatibility

Function type qualifiers

* __device__
  * Executed on GPU
  * Launched on GPU
* __host__ (optional)
  * Executed on CPU
  * Launched on CPU
* __global__
  * Executed on GPU
  * Launched on CPU

Function type qualifiers. Restrictions

* __device__ and __global__ does not support recursion*
* __device__ and __global__ should not have static variables in data section
* __device__ and __global__ have fixed number of arguments
* __global__ and __host__ should not used simultaneously
* __global__ returns void , size of arguments should be less than 256 bytes

Variable type qualifiers

* __device__
  * Resides in global memory on device
  * Has the lifetime of an application
  * Is accessed from all the threads within the grid and from the host
* __constant__
  * Resides in constant memory on device
  * Has the lifetime of an application
  * Is accessed from all the threads within the grid and from the host
* __shared__
  * Resides in the shared memory of a thread block
  * Has lifetime of the block
  * Is only accessed from all the threads within the thread block

Variable type qualifiers. Restrictions

* __shared__ variable couldn’t be initialized at declaration
* __constant__ variable could be initialized from CPU only
* Scope of __device__ and __constant__ variables - file
* __shared__ and __constant__ variables have impied static storage
* Marking elements of struct и union with those qulifiers is not allowed

Runtime configuration

* Is defined at kernel launch (__global__ function)
  * __global__ void Func(float* data);
  * Func<<<Dg, Db, Ns, S>>>(data);
* Dg — Grid size. **dim3** data type
  * Dg.x, Dg.y, Dg.z
* Db — block size. **dim3** data type
  * Db.x * Db.y * Db.z — number of threads
* Ns — additional space for shared memory. **size_t** data type. Optional. By default - 0
* S — Stream number. **cudaStream_t** data type. Optional. By default — 0.
* Kernel is executed asynchronously

Build-in variables

* **gridDim** — Grid size. dim3 data type.
* **blockIdx** — Block index. uint3 data type.
* **blockDim** — Block size. dim3 data type.
* **threadIdx** — Thread index. uint3 data type.
* uint3, dim3 - structures with 3 fields: x, y, z
* The variables are non-modifiable
* Getting address of the variables is not allowed

Example

```
// for(int i=0; i<5000; i++) c[i] = a[i] + b[i];

__global__ void my_sum(float* a, float* b,  float* c, int len){
        unsigned int index;
        index  = blockIdx.x *blockDim.x + threadIdx.x;
        if(index<len){
                c[index] = a[index] + b[index];
        }
}
dim3 BS(512);
dim3 GS(100); // dim3 GS(ceil(5000/(float)BS.x));
my_sum<<<GS, BS>>>(a, b, c, 5000);
```

Build-in vector data types

* char1, char2, char3, char4
* uchar1, uchar2, uchar3, uchar4
* short1, short2, short3, short4
* ushort1, ushort2, ushort3, ushort4
* int1, int2, int3, int4
* uint1, uint2, uint3, uint4
* long1, long2, long3, long4
* ulong1, ulong2, ulong3, ulong4
* float1, float2, float3, float4
* double1, double2
* ..
* Fields: x,y,z,w

Device initialization

* **cudaGetDeviceCount** — number of CUDA enabled devices
* **cudaGetDeviceProperties**(cudaDeviceProp*, uint) — device properties
* **cudaSetDevice**(uint) — select device. Before any routine from runtime API
  * Selected once for CUDA 3.2 and older

* To use several GPUs within a node one should write multithreading program
  * For CUDA 3.2 and older

Device properties

```
struct cudaDeviceProp{
  char       name[256];
  size_t     totalGlobalMem;
  size_t     sharedMemPerBlock;
  int        regsPerBlock;
  int        warpSize;
  size_t     memPitch;
  int        maxThreadsPerBlock;
  int        maxThreadsDim[3];
  int        maxGridSize[3];
  int        clockRate;
  size_t     totalConstMem;
  int        major;
  int        minor;
  size_t     textureAlignment;
  int        deviceOverlap;
  int        multiProcessorCount;
  int        __cudaReserved[40];
};
```

Memory allocation

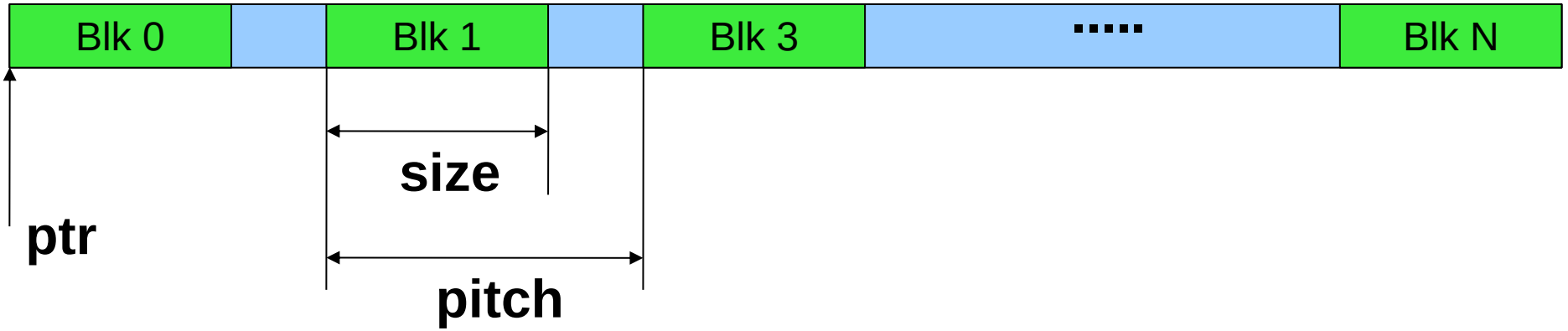
* CPU
  malloc, calloc, free, cudaMallocHost, cudaFreeHost

* GPU
  cudaMalloc, cudaMallocPitch, cudaFree

```
float* ptr;
cudaMalloc((void**)ptr, 256*sizeof(float));
....
cudaFree(ptr);
....
cudaMallocPitch((void**)ptr, &pitch, 256*sizeof(float), 256);
....
cudaFree(ptr);
```

Memory allocation

* cudaMallocPitch((void**)ptr, &pitch, size, blocks);



* cudaMallocArray(struct cudaArray **array,
  const struct cudaChannelFormatDesc* desc,
  size_t width, size_t height);
* cudaFreeArray(struct cudaArray *array);
* cudaCreateChannelDesc(int x, int y, int z, int w, enum cudaChannelFormatKind f);

Moving data to/from GPU

* cudaMemcpy(void* dst, void* src, size_t size, direction)
* direction:
  * cudaMemcpyHostToDevice
  * cudaMemcpyDeviceToHost
  * cudaMemcpyDeviceToDevice
* cudaMemcpy2D(void* dst, size_t dpitch, const void* src, size_t spitch, size_t width, size_t hight, direction)
* etc.

Atomic operation

* atomicAdd, atomicSub, atomicExch, atomicMin, atomicInc, atomicDec, atomicCAS, atomicOr, atomicAnd, atomicXor, etc
* Compute compatibility 1.1+
  * Integer atomic functions operating on 32-bit words
* Compute compatibility 2.x+
  * Floating-point atomic functions operating on 32-bit words

Math functions

* Some math functions can be used in both host and device; some functions can be used in device code only
* ADD, MUL functions are IEEE-compatible.
* There are ‘fast’ functions. Precision could be differ. See Programming guide.

* `sin(), sinf(), __sinf()`

Scenario

1. Select GPU
2. Allocate memory for input/output data
3. Launch kernel(s)
4. Copy result back to the host.
5. If required, repeat 1-4
6. Free memory