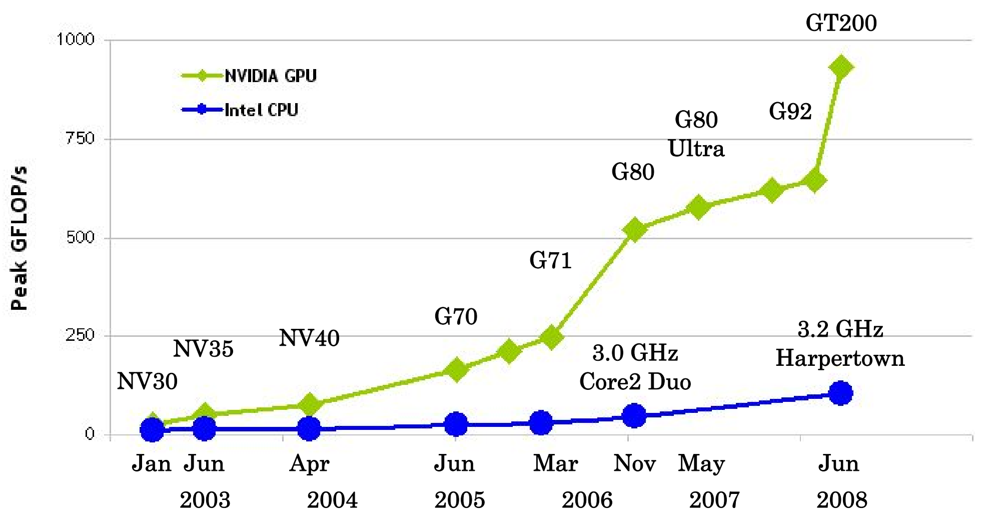
Joined Advanced Student School (JASS) 2009
March 29 - April 7, 2009
St. Petersburg, Russia

GPGPU: HIGH-PERFORMANCE COMPUTING

Dmitry Puzyrev
St. Petersburg State University
Faculty of Physics
Department of Computational Physics

In recent years the application of graphics processing units to general purpose computing becomes widely developed. GPGPU, which stands for General-purpose computing on Graphics Processing Units, makes its way into the fields of computations, traditionally associated with and handled on CPUs or clusters of CPUs. The breakthrough in the area of GPU computing was caused by the introduction of programmable stages and higher precision arithmetics on rendering pipelines, allowing to perform stream processing of non-graphics data.

Let's understand, what makes GPUs effective in high-performance computing. GPUs are built for parallel processing of data, and are highly effective in data parallel tasks. High amount of computing units (GPUs have in the range of 128-800 ALUs, compared to 4 ALUs on a typical quad-core) allows computation power of GPU to exceed that of CPU by up to 10 times for high-end models, while high-end GPUs cost much less than CPUs (see Fig. 1). Memory bandwidth, essential to many applications, is 100+ GB/s, compared to ≈10-20 GB/s for CPUs.



GT200 = GeForce GTX 280	G71 = GeForce 7900 GTX	NV35 = GeForce FX 5950 Ultra
G92 = GeForce 9800 GTX	G70 = GeForce 7800 GTX	NV30 = GeForce FX 5800
G80 = GeForce 8800 GTX	NV40 = GeForce 6800 Ultra	

*Fig. 1. Comparison of computational power:
floating-point operations per second for the GPU and CPU (by NVIDIA)*

Such computational power can now be applied in different areas, including scientific computing, signal and image processing, and, of course, computer graphics itself (including non-traditional rendering algorithms, e.g. ray tracing). Scientific applications of high-performance GPGPU include molecular dynamics, astrophysics, geophysics, quantum chemistry, neural networks. Fig. 2 illustrates the effectiveness (speedup) of GPGPU in several fields.

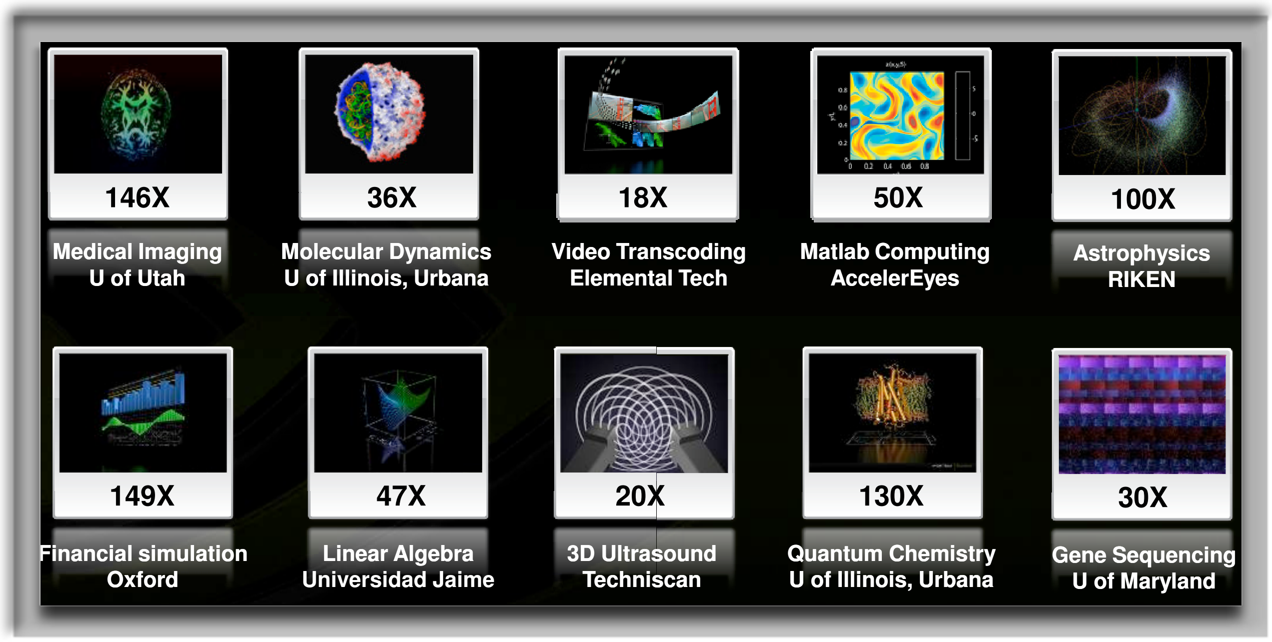

146X
Medical Imaging U of Utah
36X
Molecular Dynamics U of Illinois, Urbana
18X
Video Transcoding Elemental Tech
50X
Matlab Computing AccelerEyes
100X
Astrophysics RIKEN
149X
Financial simulation Oxford
47X
Linear Algebra Universidad Jaime
20X
3D Ultrasound Techniscan
130X
Quantum Chemistry U of Illinois, Urbana
30X
Gene Sequencing U of Maryland

Fig. 2. Speedup by use of GPGPU in scientific applications (by NVIDIA)

GPGPU computing can be performed on various hardware, including virtually all modern GPUs. NVIDIA desktop GPUs (9x or GTX series) and modern AMD GPUs support general-purpose computing. Both NVIDIA and AMD have their own dedicated high-performance GPGPU solutions, which are NVIDIA Tesla and AMD FireStream.

Of course, GPU architecture is highly specific. Basically, GPU devotes much more transistors to data processing rather than data caching and flow control. Fig. 3 roughly illustrates the CPU and GPU architecture specifics.

Hardware specifics affect the programming model. The basics of GPGPU programming model are:

- Small program (called kernel) works on many data elements
- Each data element is processed concurrently
- Communication is effective only inside one execution unit.

Two slightly different models are used on GPUs: SIMD (Single instruction, multiple data) and SPMD (Single program, multiple data).

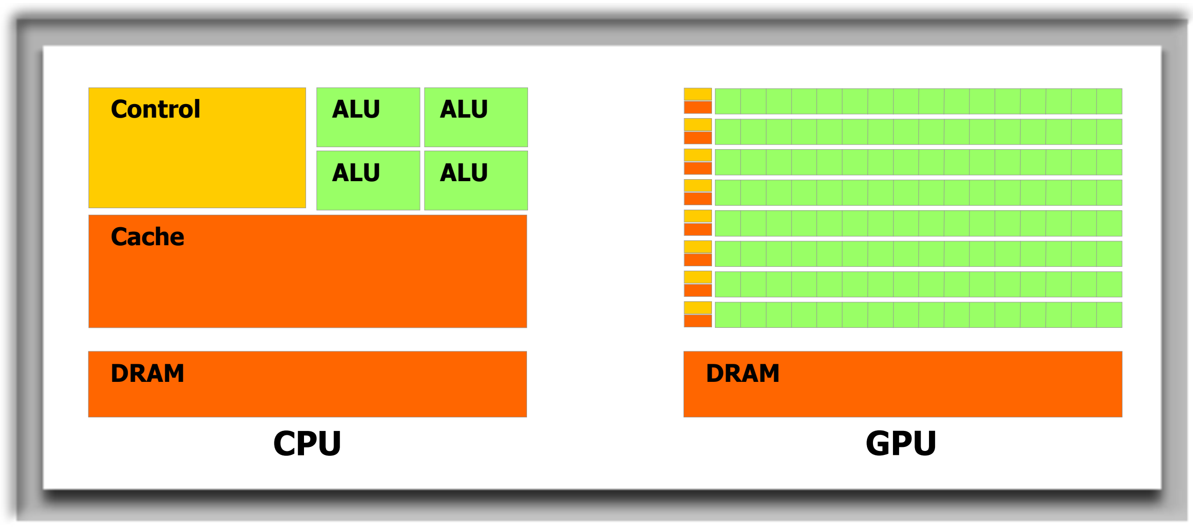


Fig. 3. Schematic comparison of CPU and GPU architecture

The most popular instrument of general-purpose GPU computing is NVIDIA's GPGPU implementation, that is called NVIDIA CUDA. CUDA is positioned as general purpose parallel computing architecture and works with all modern NVIDIA GPUs. It uses C as high-level programming language, though other languages will be supported in the future, as seen on Fig. 4.

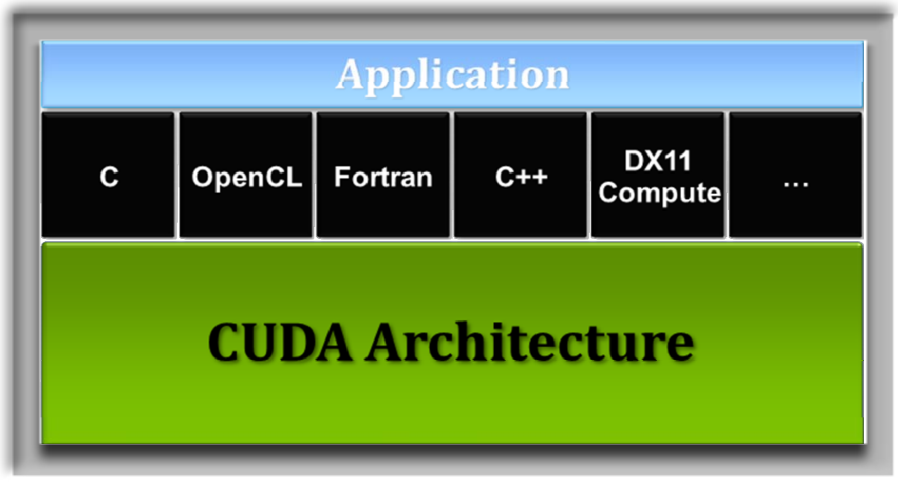


Fig. 4. NVIDIA CUDA architecture

One of the basics for CUDA is the concept of *kernels*. These are C functions, that are executed N times in parallel with specific <<<>>> syntax in main functions. Fig. 5 contains the basic example of CUDA code with kernel and main function.

```
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Kernel invocation
    vecAdd<<<1, N>>>(A, B, C);
}
```

Fig. 5. Kernel: this code adds two vectors A and B of size N

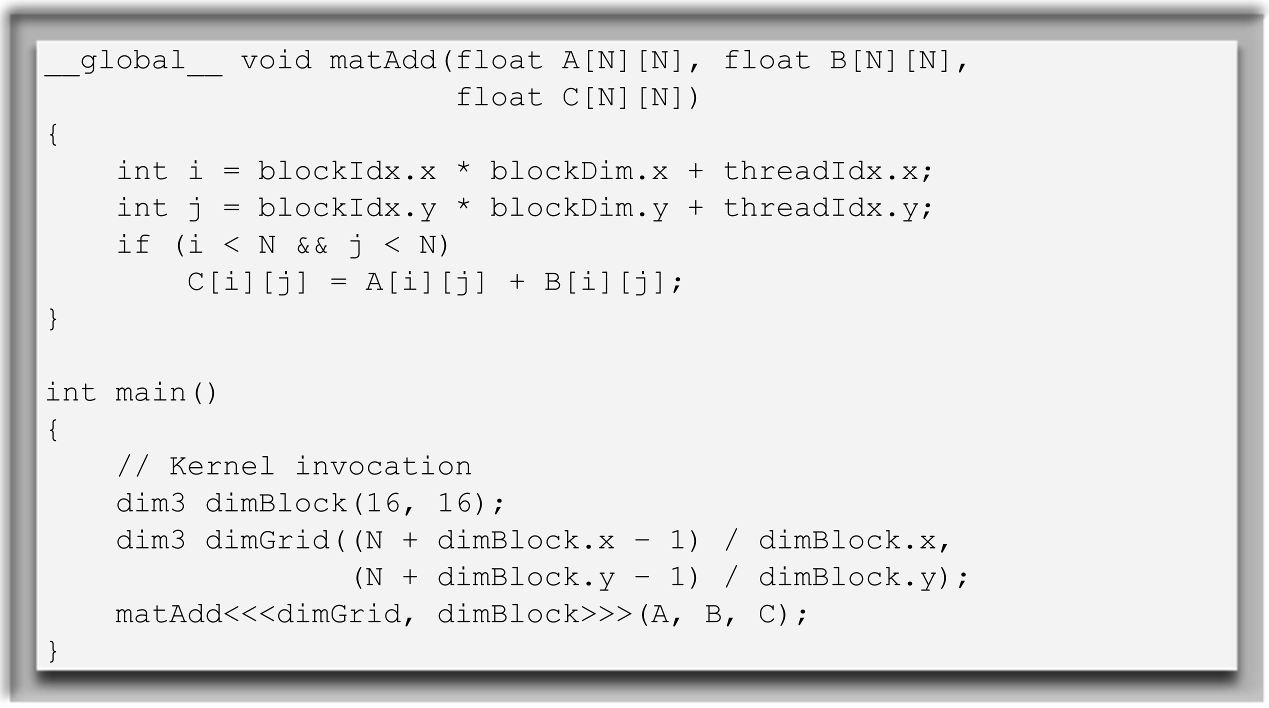
The next basic concept of CUDA is *thread hierarchy.* One kernel is executed in one *thread*, and threads are combined in *thread blocks.* Next figure shows an example of matrix addition using thread blocks.

```
__global__ void matAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    // Kernel invocation
    dim3 dimBlock(N, N);
    matAdd<<<1, dimBlock>>>(A, B, C);
}
```

*Fig. 6. Thread blocks: this code adds two matrices A and B of size N*N*

Threads within a block have shared memory and can synchronize. On current GPUs, a thread block may contain up to 512 threads. A kernel can be executed by multiple equally shaped thread blocks put in a *grid*. Thread blocks in a grid are required to execute independently. Fig. 7 shows an example of matrix addition on a grid. Fig. 8 shows the full hierarchy of threads.



```
__global__ void matAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    // Kernel invocation
    dim3 dimBlock(16, 16);
    dim3 dimGrid((N + dimBlock.x – 1) / dimBlock.x,
                 (N + dimBlock.y – 1) / dimBlock.y);
    matAdd<<<dimGrid, dimBlock>>>(A, B, C);
}
```

*Fig. 7. Grid: this code adds two matrices A and B of size N*N*

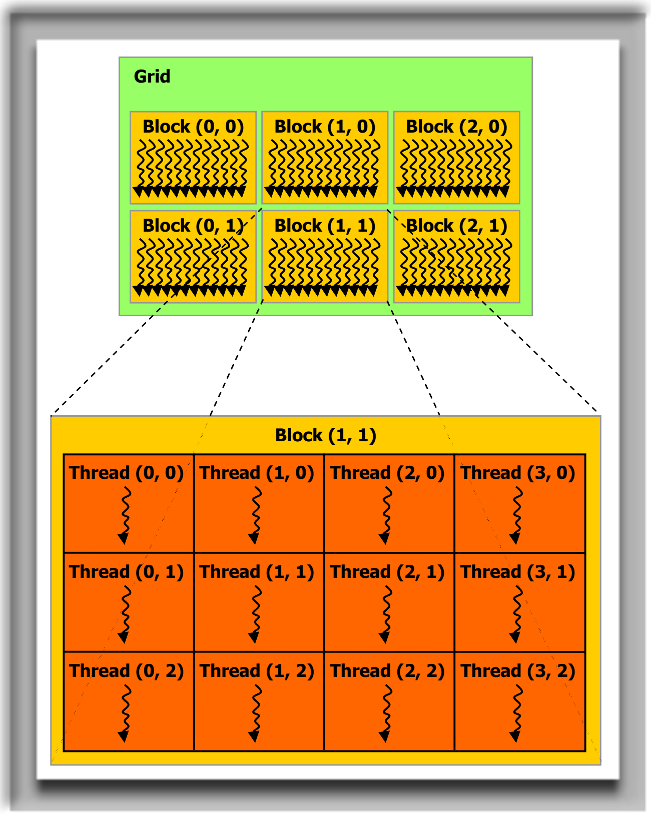


Fig. 8. Full thread hierarchy

Memory hierarchy is another complicated part of CUDA. As shown on Fig. 9, multiple memory spaces are present, including additional specialized memory spaces: constant and texture memory. Different memory usage strategies are used for different applications. Memory usage is usually the bottleneck of GPGPU applications.

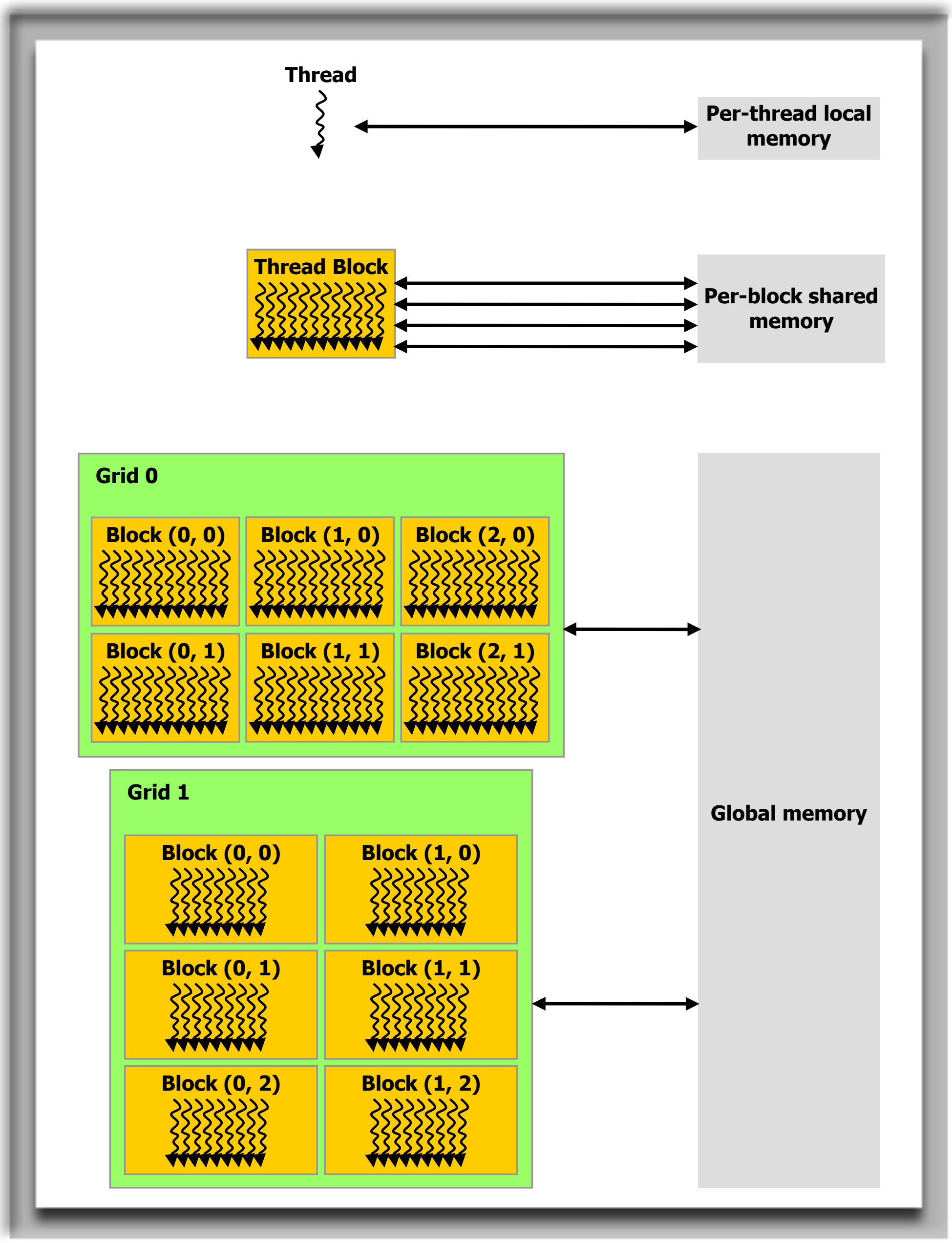


Fig. 9. Memory hierarchy

The *host* and *device* model controls the execution of CUDA program. CUDA threads are executed execute on a physically separate device that operates as a coprocessor to the host running the C program. Both the host and the device maintain their own DRAM, referred to as host memory and device memory, and CUDA runtime manages data transfer.

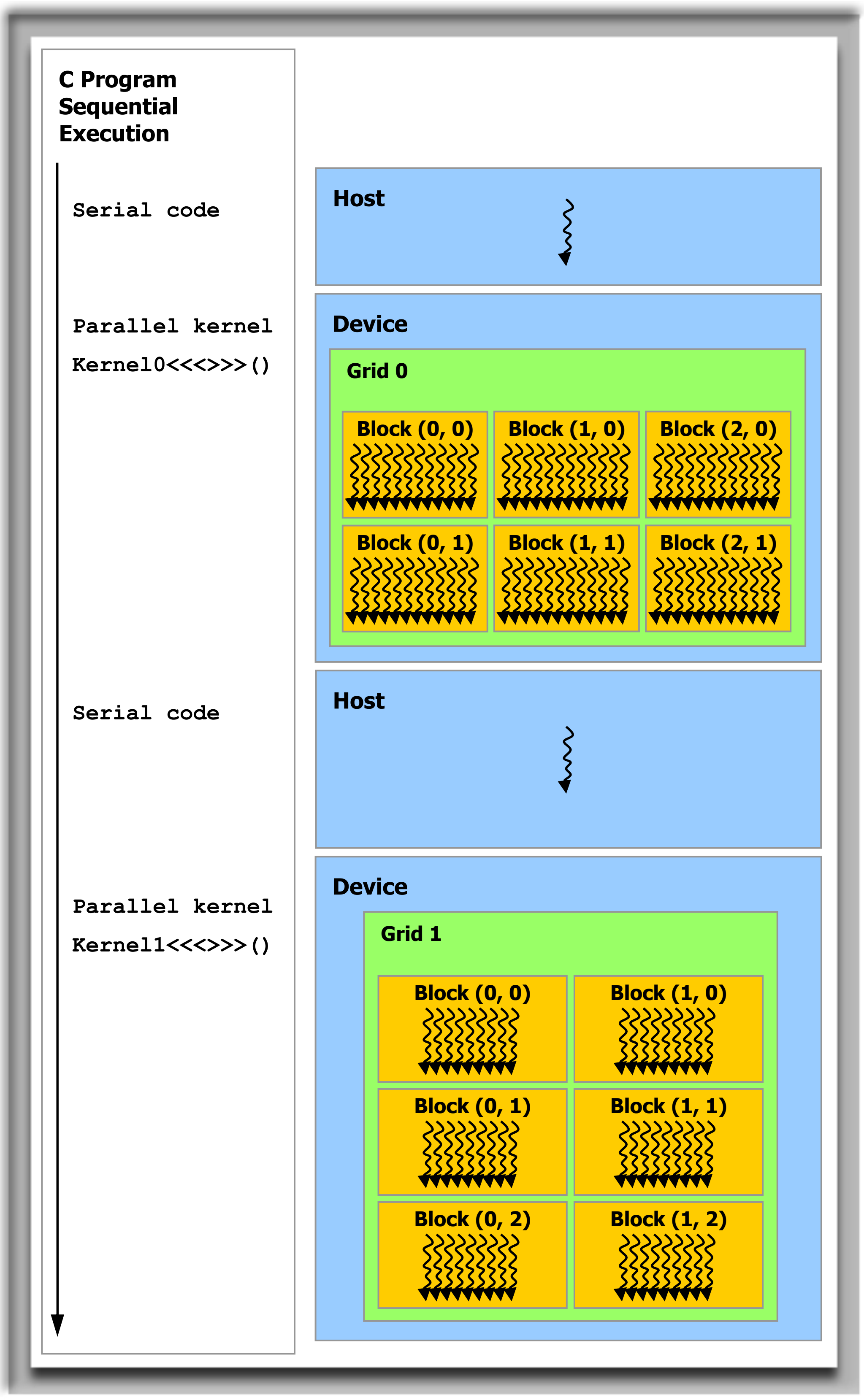


Fig. 10. Host and device: CUDA program execution

CUDA allows to implement various libraries of functions, that essential for scientific high-performance computing. There is an efficient implementation of FFT on CUDA, as well as an implementation of BLAS, called CUBLAS. Unfortunately, CUBLAS doesn't yet include all BLAS functions. CUDA-based plugins for MATLAB exists. LAPACK libraries for CUDA are under development. CUDA allows to perform efficient operations on dense matrices, sparse solvers are still under development.

CUDA still has many limitations. Of course, some of them are fundamental and come from specific architecture and data-parallel programming model. But other limitations should definitely be fixed in the future. For example, double precision has no deviations from IEEE 754 standard, but single precision is not standard. Double precision is supported only by the last generation of NVIDIA GPUs. CUDA still lacks advanced profiler. Emulation on CPU is slow and often results, which are different from that on GPU. Another drawback is that CUDA works only on NVIDIA GPUs.

Alternative implementation of GPGPU architecture is Stream Computing by AMD, which is based on Brook programming language. Brook works on both AMD and NVIDIA GPUs, Brook+ is AMD hardware optimized version. Brook is faster in some applications and has better support of double precision.

There are two main future directions in GPGPU computing. The first focuses on creation of new hardware, which should be more suitable for general-purpose computing. The best example is *Larrabee* - Intel's upcoming discrete GPU. It will compete with both GPUs and high-performance computing. Larrabee has a hybrid architecture: 16-32 simple x86 cores with SIMD vector units and per-core L1/L2 cache, without fixed function hardware, except for texturing units.

Programming model for Larrabee will be task-parallel on core level and data-parallel on vector units inside the core. One significant feature is that Larrabee cores can submit work to itself without the host. Larrabee will use Intel C/C++ compiler and benefit from all its features. Larrabee gathers much praise from Intel and huge criticism from GPU manufacturers.

Another direction is to create a standardized API for programming on different architectures. *OpenCL,* which stands for Open Computing Language, is a framework for programming on CPUs, GPUs and other processors. It was proposed by Apple and developed by Khronos Group, and has full support from both AMD and NVIDIA. OpenCL is likely to be introduced with Mac OS X 10.6.

For the conclusion, let's make a statement: GPGPU is a developed branch of computing. NVIDIA CUDA already allows us to utilize the power of GPUs in convenient way.

Data parallel programming model is effective in many applications, but GPGPU could become more flexible, supporting more programming languages and programming concepts.

A fusion between many-core CPUs and GPUs is a promising direction, and this direction is explored by Intel.

Standardized API for GPGPU is highly anticipated, and upcoming OpenCL is the main candidate for this API.

REFERENCES

NVIDIA CUDA Programming Guide 2.1

Various **NVIDIA CUDA** presentations