GPGPU

High-performance computing

Dmitry Puzyrev, SPbSU

for JASS ’09

What is GPGPU?

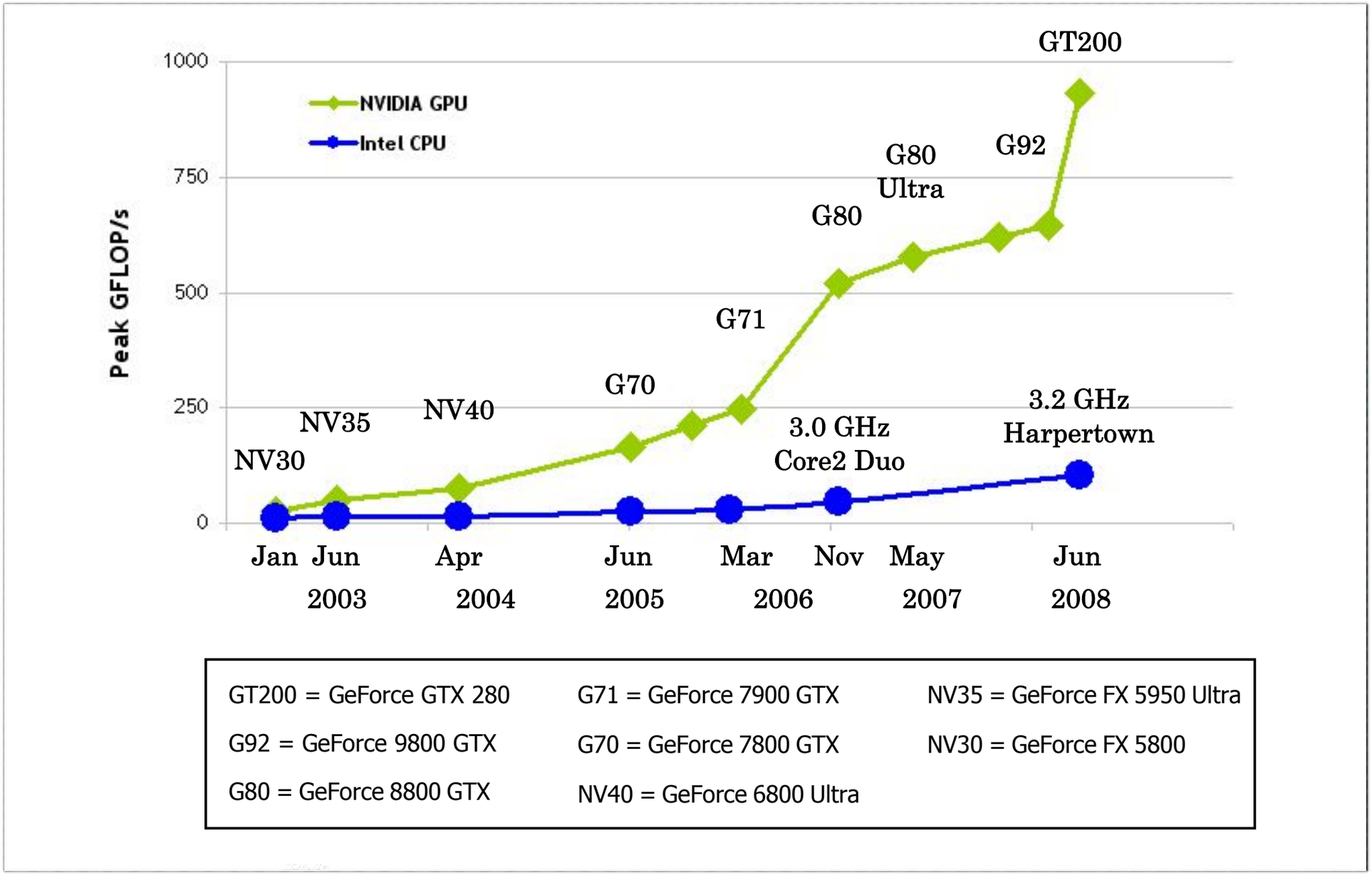
- General-purpose computing on Graphics Processing Units
- Applications traditionally handled by the CPU
- Programmable stages and higher precision arithmetics on rendering pipelines
- Stream processing of non-graphics data

Why GPGPU can be effective?

- GPUs are built for parallel processing
- 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)
- High memory bandwidth (100+ GB/s, compared to ≈10-20 GB/s for CPUs)

Computational power of GPU

Floating-point operations per second for the GPU and CPU:



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	

Application areas:

- High-performance computing
- Signal processing
- Image processing
- Computer graphics:
  - Raytracing
  - Rendering tasks

High-performance computing on GPGPU: Scientific applications

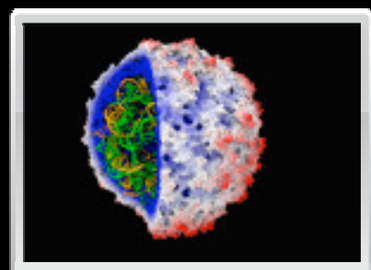
- Molecular dynamics
- Astrophysics
- Geophysics
- Quantum chemistry
- Neural networks

etc.

Speedup by GPU:

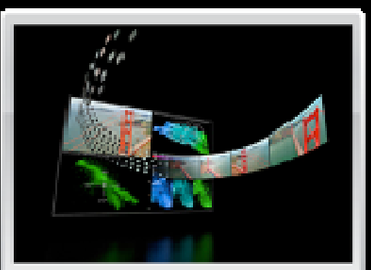
146X

Medical Imaging
U of Utah



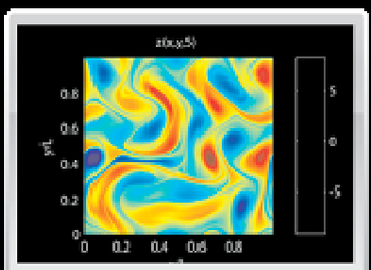
36X

Molecular Dynamics
U of Illinois, Urbana



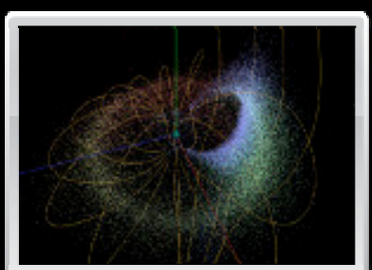
18X

Video Transcoding
Elemental Tech



50X

Matlab Computing
AccelerEyes

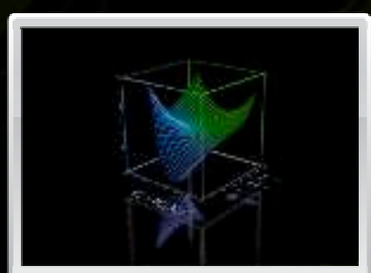


100X

Astrophysics
RIKEN

149X

Financial simulation
Oxford



47X

Linear Algebra
Universidad Jaime

20X

3D Ultrasound
Techniscan

130X

Quantum Chemistry
U of Illinois, Urbana

30X

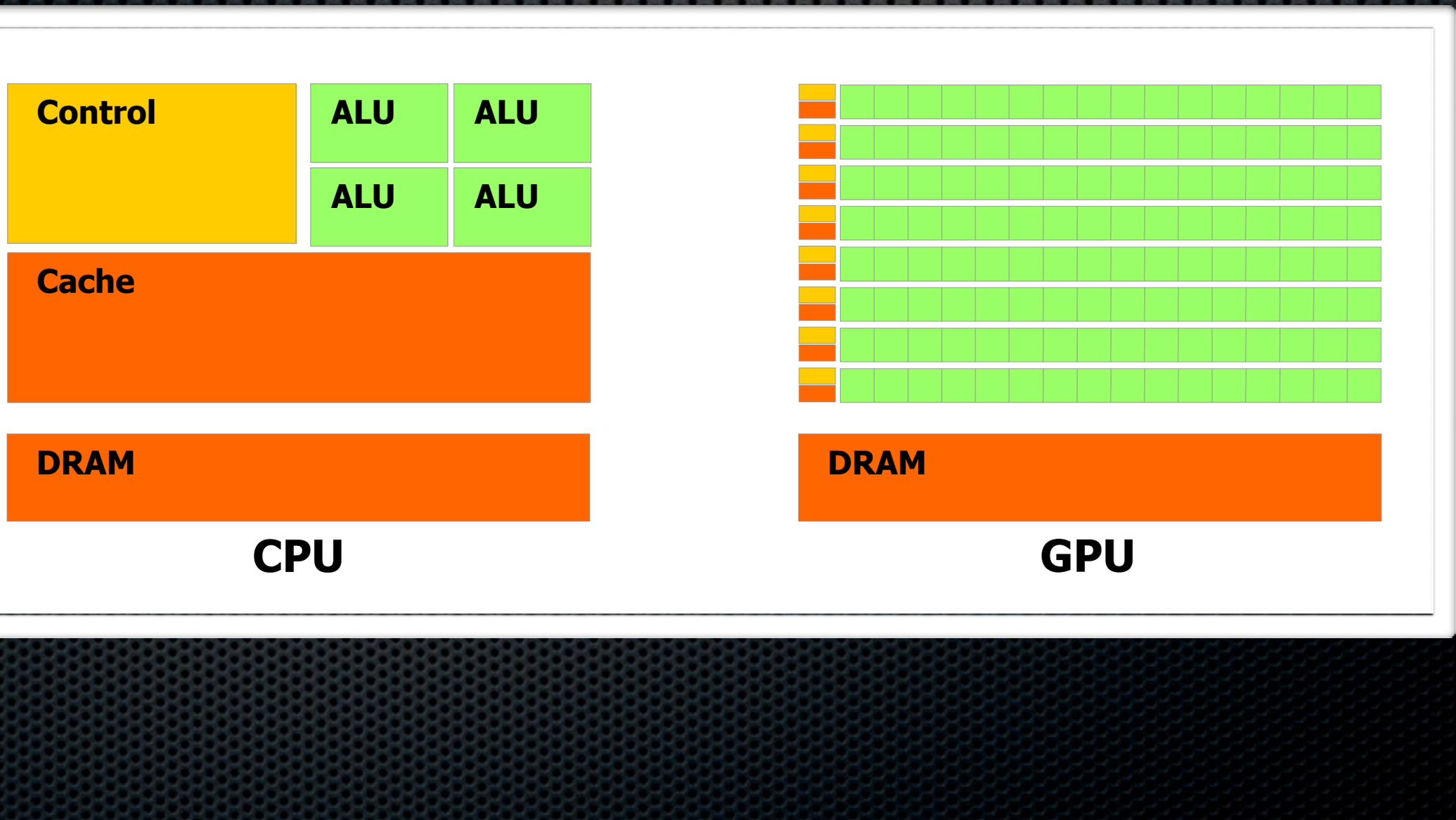
Gene Sequencing
U of Maryland

GPGPU hardware:

- NVIDIA graphic processors:
  - from desktop GPUs (9x or GTX series) to dedicated Tesla and Quadro
- AMD graphic processors
  - AMD FireStream is a dedicated solution
- Cell by Sony, Toshiba and IBM

Specifics of GPU architecture

GPU devotes much more transistors to data processing rather then data caching and flow control

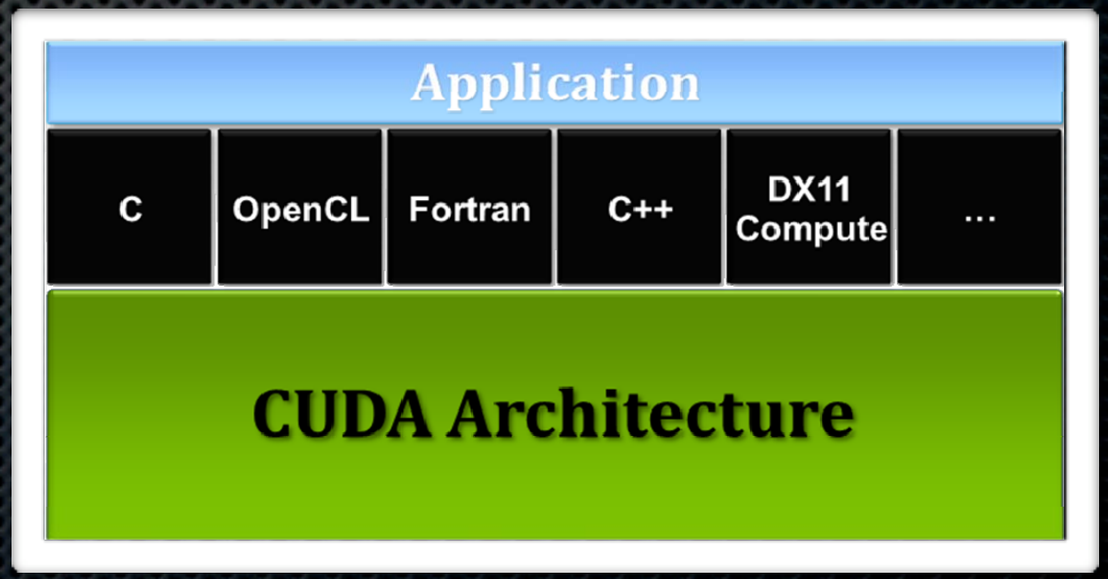


Programming model:

- Small program (called kernel) works on many data elements
- Each data element is processed concurrently
- Communication is effective only inside one execution unit
- Different models:
  - **SIMD** (Single instruction, multiple data)
  - **SPMD** (Single program, multiple data)

GPGPU implementation: **NVIDIA CUDA**

- General purpose parallel computing architecture
- Works with all modern NVIDIA GPUs
- Uses C as a high-level programming language
- Other languages will be supported in the future



Basic CUDA concepts: Kernels

- C functions, called *kernels*, that are executed N times in parallel with <<< >>> syntax in 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);
}
```

- This code adds two vectors A and B of size N

Basic CUDA concepts: Thread Hierarchy

- One *kernel* is executed in one *thread*, and threads are combined in *thread blocks*

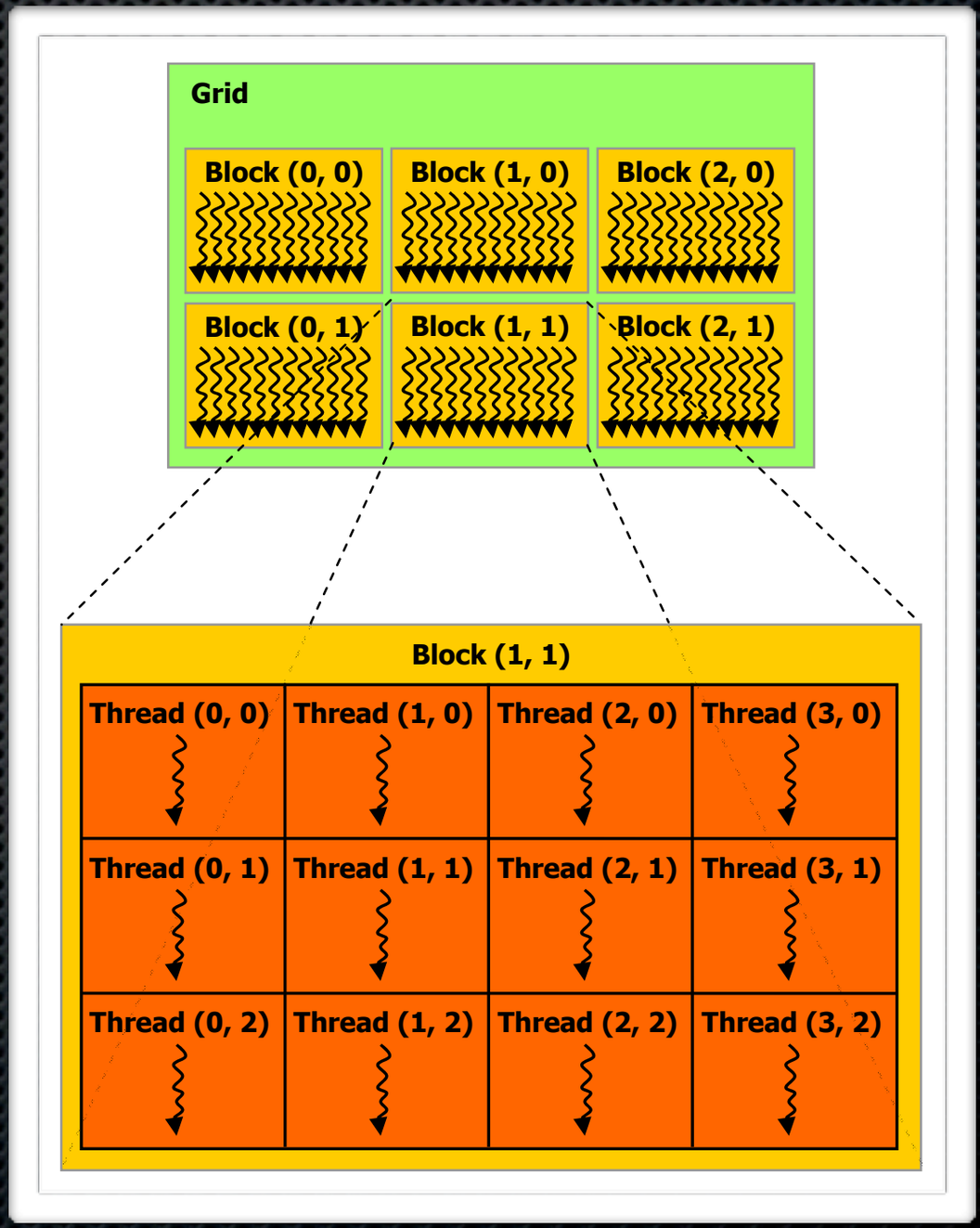
```
__global__ void matAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

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

- This code adds two matrices A and B of size NxN

Threads and thread blocks

- 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 and grid

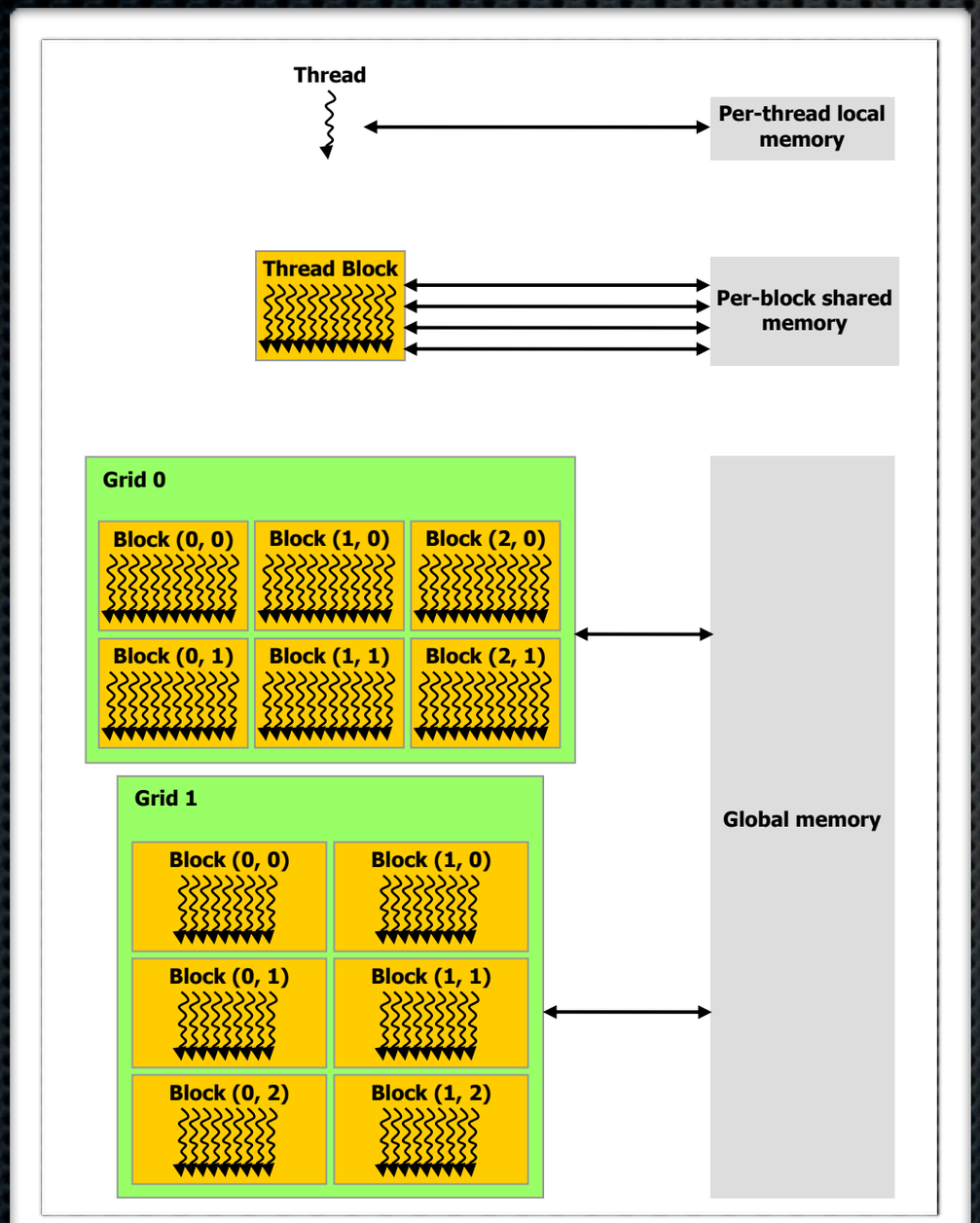
- Matrix addition on a *grid*

```
__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);
}
```

- *Thread blocks* in a *grid* are required to execute independently

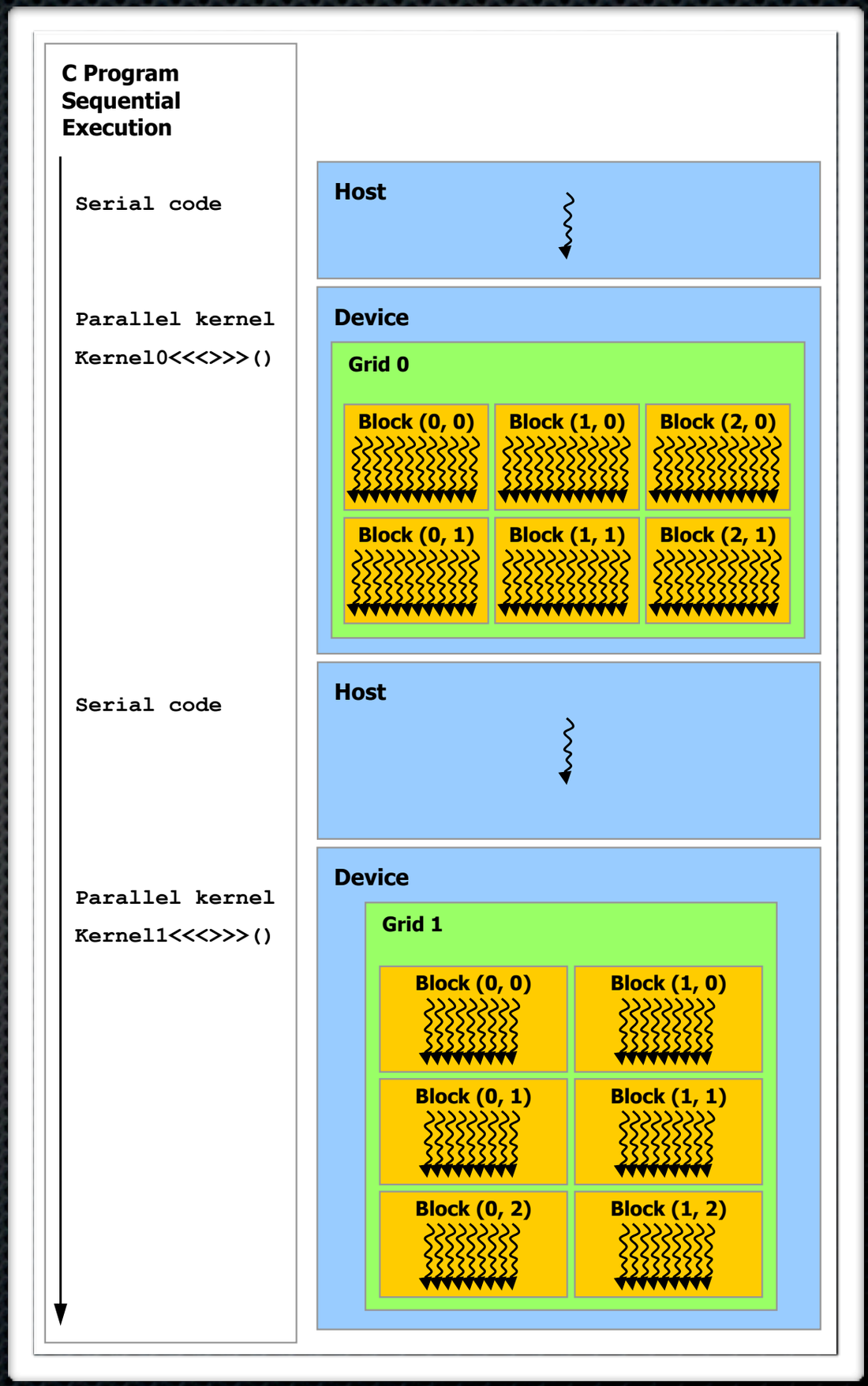
Basic CUDA concepts:
Memory Hierarchy



- Multiple memory spaces
- Additional memory spaces: constant and texture memory
- Different memory usage strategies
- Memory usage: bottleneck of GPGPU applications

Basic CUDA concepts: Host and Device

- CUDA threads may 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.



CUDA features

- Efficient implementation of FFT on CUDA
- CUDA implementation of BLAS, called CUBLAS
  - Subset of BLAS functions
- CUDA-based plugins for MATLAB
- LAPACK libraries for CUDA are under development

- Efficient dense linear algebra, sparse solvers are under development

CUDA limitations

- Fundamental advantages and limitations come from specific architecture and data-parallel programming model
- Double precision has no deviations from IEEE 754 standard, single precision is not standard
- Still lacks advanced profiler
- Emulation on CPU is slow and often gives different results
- Works only on NVIDIA GPUs

Alternative implementation: Brook

- Alternative architecture is built by AMD, called Stream Computing, based on Brook language
- Brook works on both AMD and NVIDIA GPUs, Brook+ is AMD hardware optimized version
- Faster in some applications, better support of double precision
- Apparently, AMD doesn’t actively compete to NVIDIA in high-performance GPGPU market at the moment

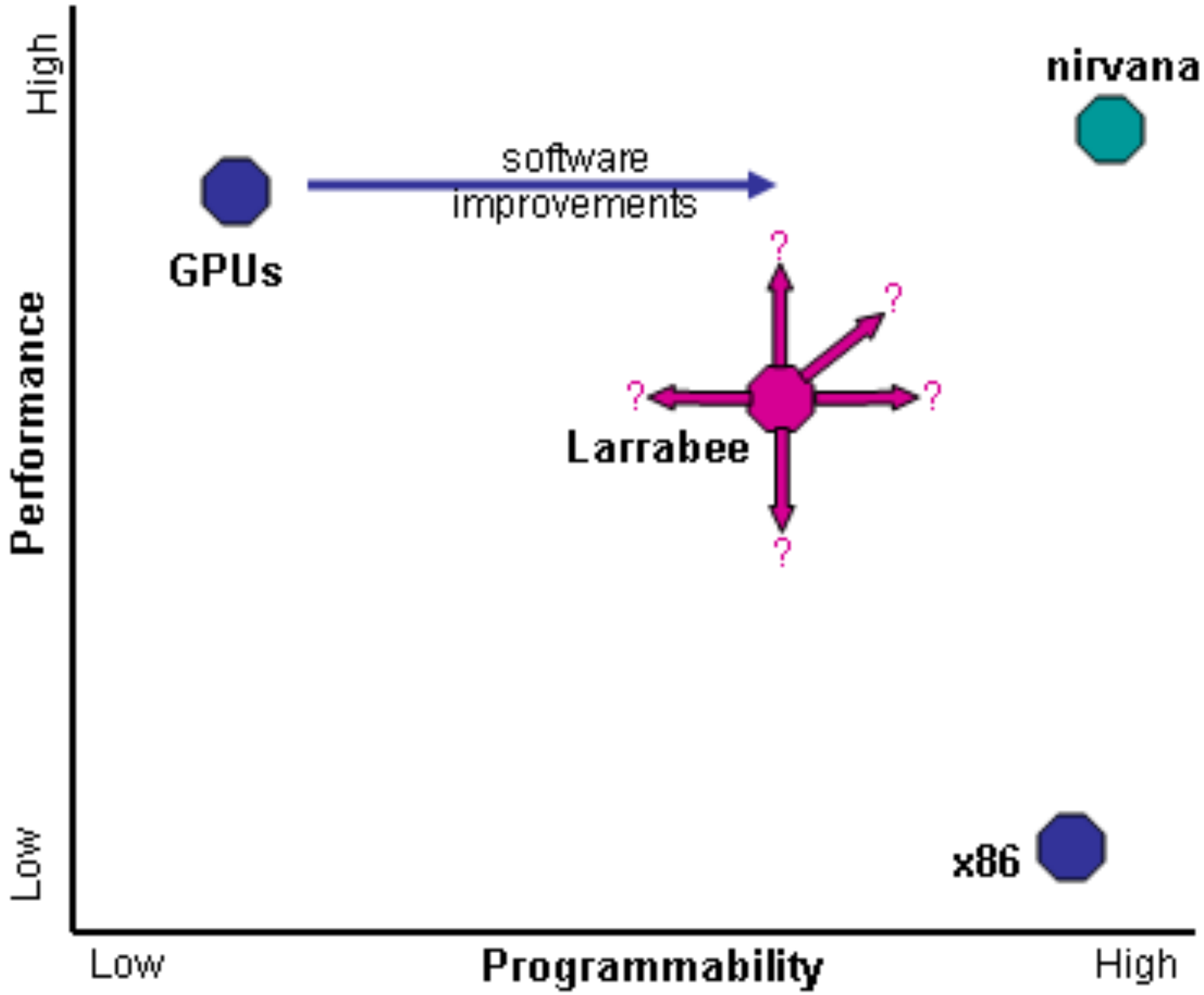
Future directions: Larrabee

- Larrabee - Intel's upcoming discrete GPU
- Compete with both GPUs and high-performance computing
- Hybrid architecture: 16-32 simple x86 cores with SIMD vector units and per-core L1/L2 cache
- No fixed function hardware, except for texturing units

Larrabee: Programming concepts

- Task-parallel on core level
- Data-parallel on vector units inside the core
- Cores can submit work to itself without the host
- Benefits from Intel C/C++ compiler

Much praise from Intel and much criticism from GPU manufacturers


nirvana
software
improvements
GPUs
?
?
?
?
?
Larrabee
x86
Performance
High
Low
Low
Programmability
High

Future directions: OpenCL

- OpenCL - a framework for programming on CPUs, GPUs and other processors
- Proposed by Apple and developed by Khronos Group
- Full support from both AMD and NVIDIA
- Will be introduced with Mac OS X 10.6

Conclusion

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

A fusion between many-core CPUs and GPUs is a promising direction

Standardized API for GPGPU is highly anticipated