NVIDIA®

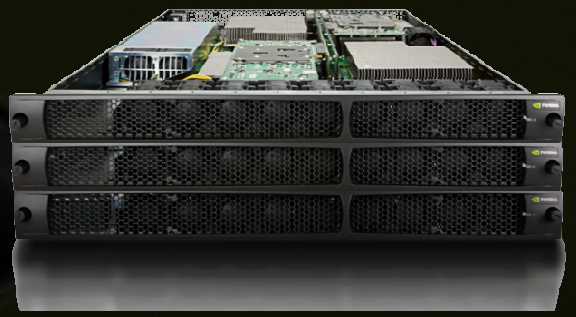
GPU

Ben de Waal

Summer 2008

Agenda

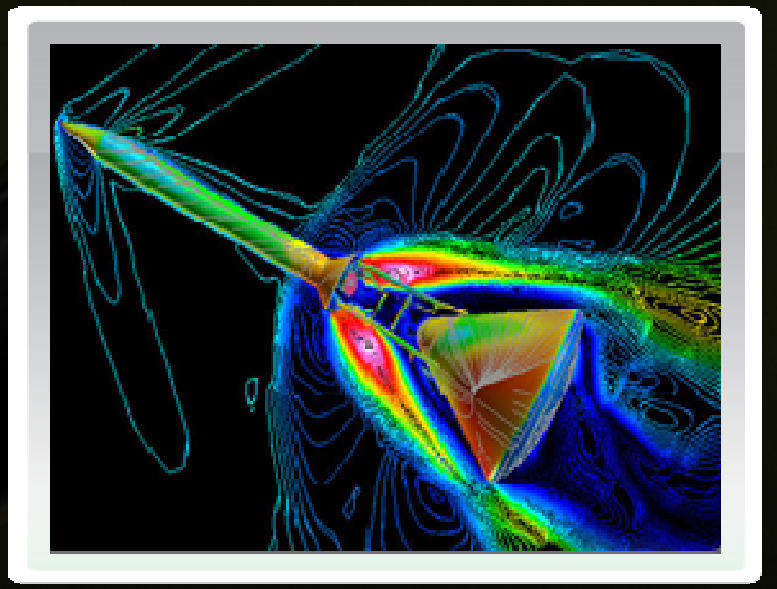
- Quick Roadmap
- A few observations
- And a few positions



GPUs are Great at Graphics

GPUs are Great at Other Things!

- An expanding trend over the last few years
- Successful applications in many areas
  - Computational geometry, biology, chemistry, physics, finance…
  - Computer vision
  - Database management
  - Signal processing
  - Physics simulation
  - …

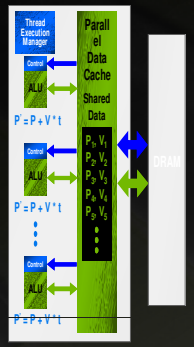


C for the GPU

CUDA & GPU Computing

NVIDIA

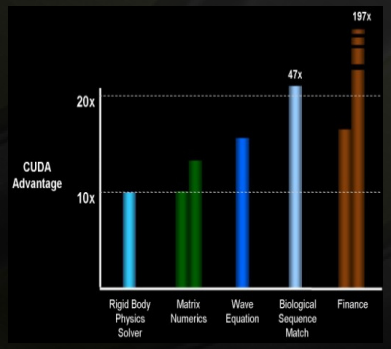
New Architecture for Computing



Standard C Programming

```
dim3  DimGrid(100, 50);   // 5000 thread blocks
dim3  DimBlock(4, 8, 8);  // 256 threads per block
size_t SharedMemBytes = 64; // 64 bytes of shared memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

CUDA

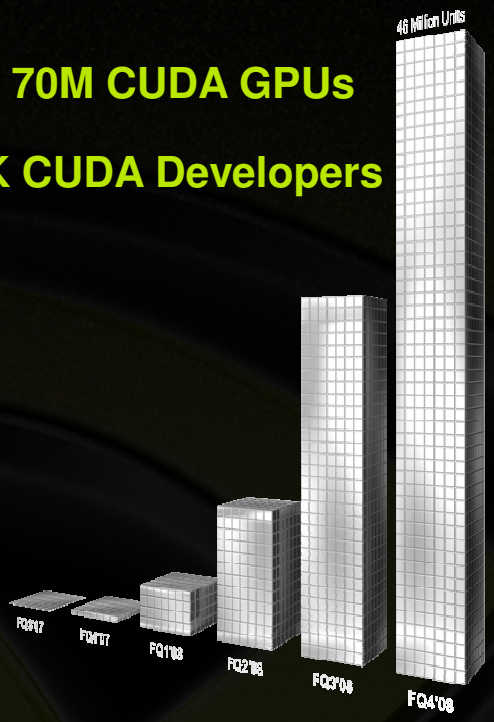


Unprecedented Performance

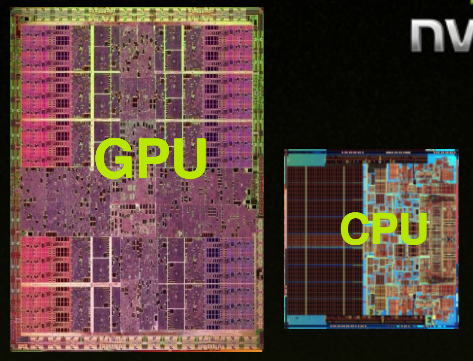
New Applications

70M CUDA GPUs

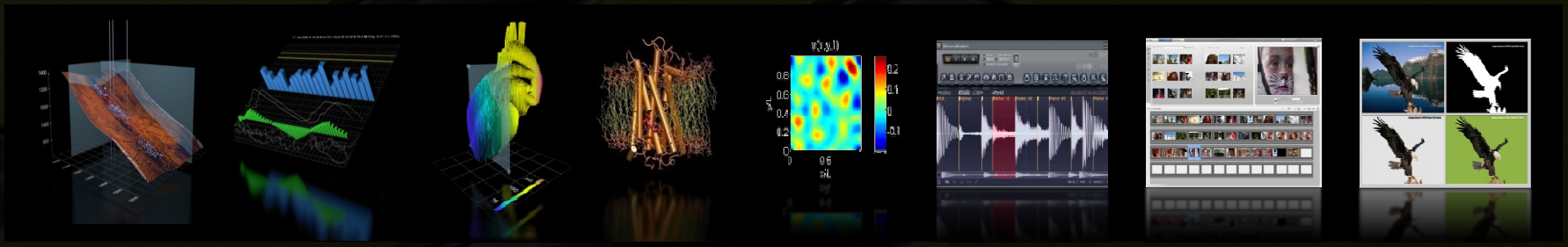
60K CUDA Developers



CUDA

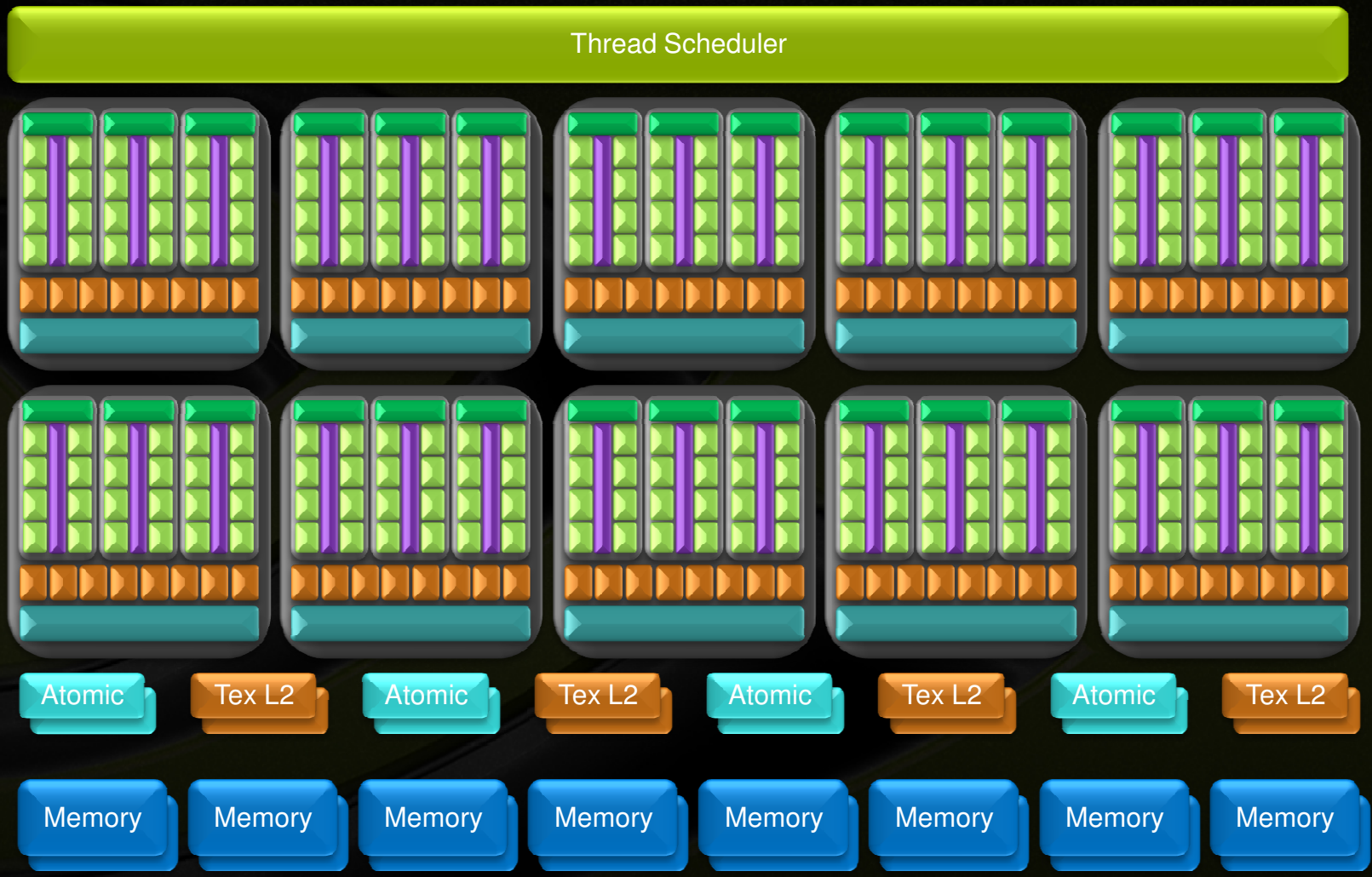


Heterogeneous Computing



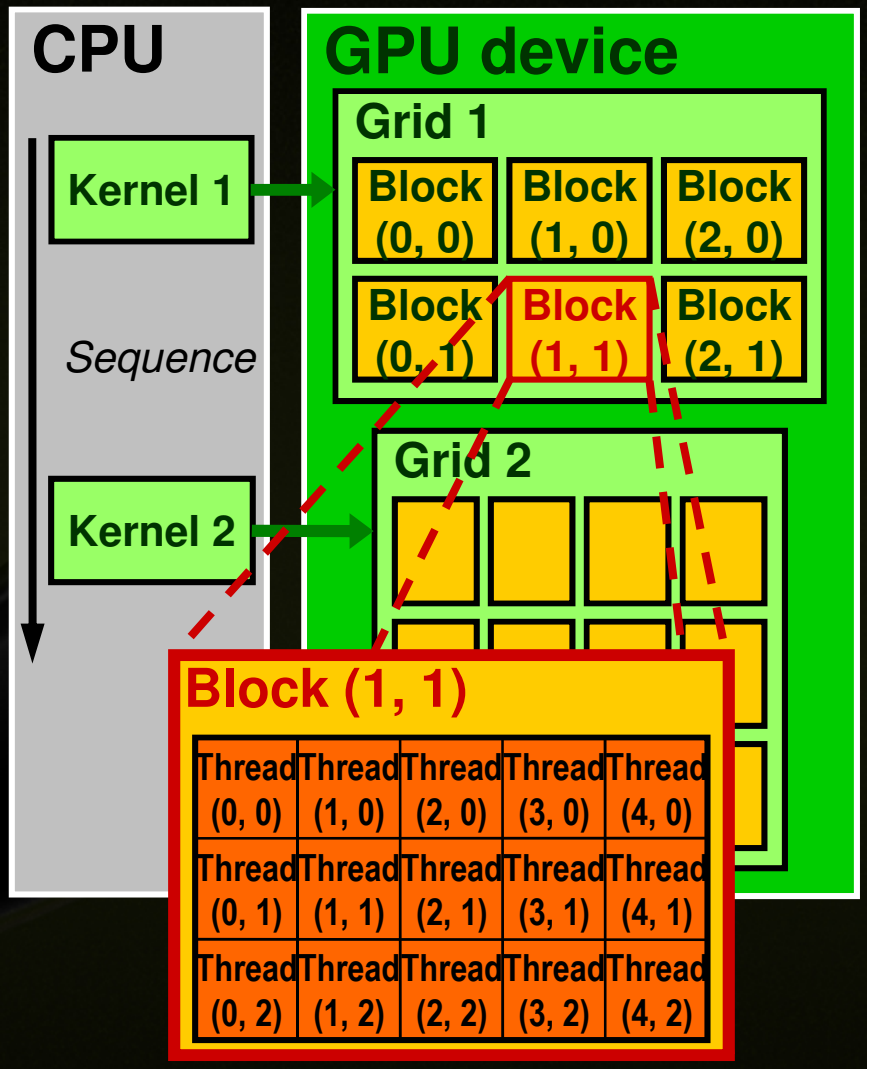
Oil & Gas | **Finance** | **Medical** | **Biophysics** | **Numerics** | **Audio** | **Video** | **Imaging**

GeForce GTX 280 Parallel Computing Architecture

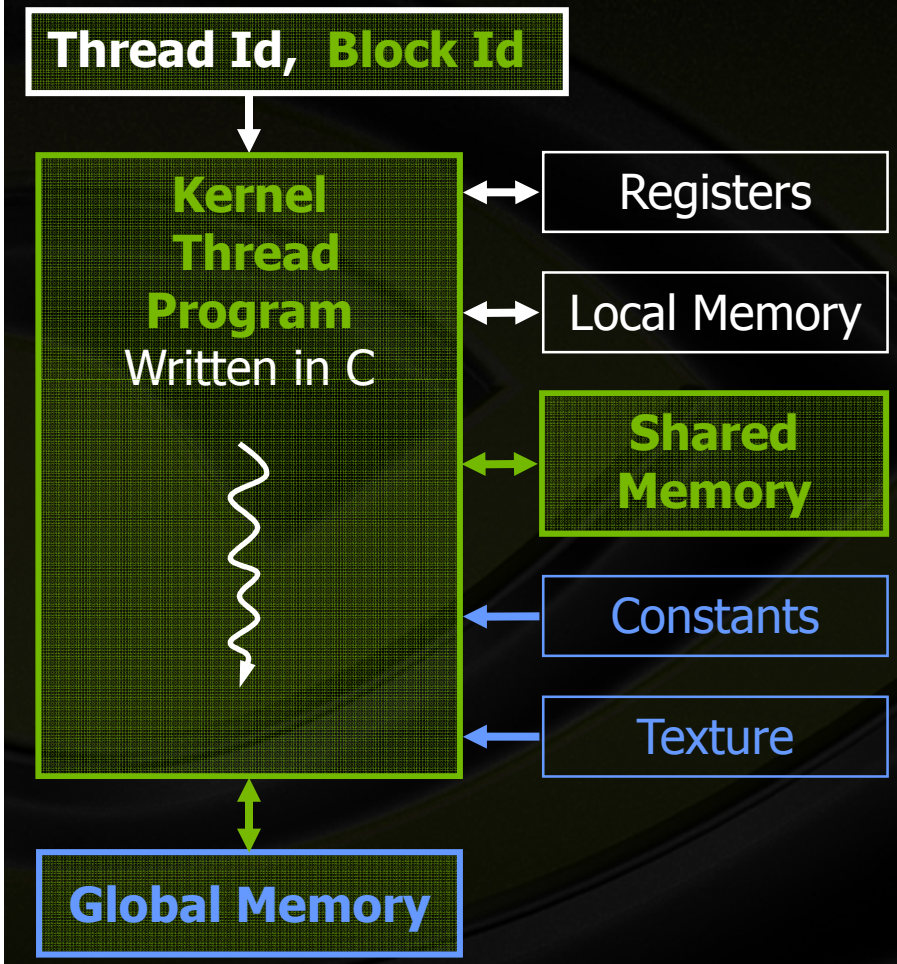


CUDA Terminology: Grids, Blocks, and Threads

- Programmer partitions problem into a sequence of kernels.
- A kernel executes as a grid of thread blocks
- A thread block is an array of threads that can cooperate
- Threads within the same block synchronize and share data in Shared Memory
- Execute thread blocks on multithreaded multiprocessor SM cores



CUDA Programming Model: Thread Memory Spaces



- Each kernel thread can read:
  - Thread Id — per thread
  - Block Id — per block
  - Constants — per grid
  - Texture — per grid
- Each thread can read and write:
  - Registers — per thread
  - Local memory — per thread
  - Shared memory — per block
  - Global memory — per grid
- Host CPU can read and write:
  - Constants — per grid
  - Texture — per grid
  - Global memory — per grid

Trends and Observations

- Cores generally doubles per family
  - Low end has substantially less cores than high end
  - Ranges from 8 to 100s
- Memory hierarchy will likely remain
- Evolving
  - Processor expressiveness
  - Easy of programming
  - Reducing performance cliffs
  - Hierarchical scheduling & partitioning
  - Nested parallelism
- Heterogeneous computing
  - Algorithms vary
  - Run them on the most suitable processor

Autotuning – Super Languages

- One Possible extreme outcome:

  People program in an expressive enough language that maps fairly cleanly onto the installed base of processors
  - Programmer driven
  - Just very simple machine translation needed

  As an example, CUDA's programming paradigm also scales with CPU cores
  - Data parallel
  - Memory hierarchy is explicit
  - i.e. It reflects an architectural superset of several different designs

Heterogeneous Tuning Space

- Cache hit architectures
  - Like traditional CPUs
  - Thread driven execution
  - NUMA / Cost of global coherence
  - Cache cliffs (hits, misses, aliasing, etc.)
  - Scalar / Vector (SIMD)
- Cache miss architectures
  - Like many GPUs
  - Data driven execution
  - Wide range of cores
  - NUMA / sometimes no global coherence
  - Memory technology exposure (banks, etc.)
  - Vector / Scalar

Autotuning – Really Smart Code

- Another extreme outcome:

Genetic programming style autotuners

- Evolves optimal code for any (local) architecture
- Potential to find a diamond in the state of Texas
- Somehow still generalize
- Good news: It's parallelizable!

Detour: Circuit Synthesis

- Similar Problem
- Remarkable success
- Remarkable exploitation

Genetic Programming III, Koza, John R, et al, 1999
Chapter 25

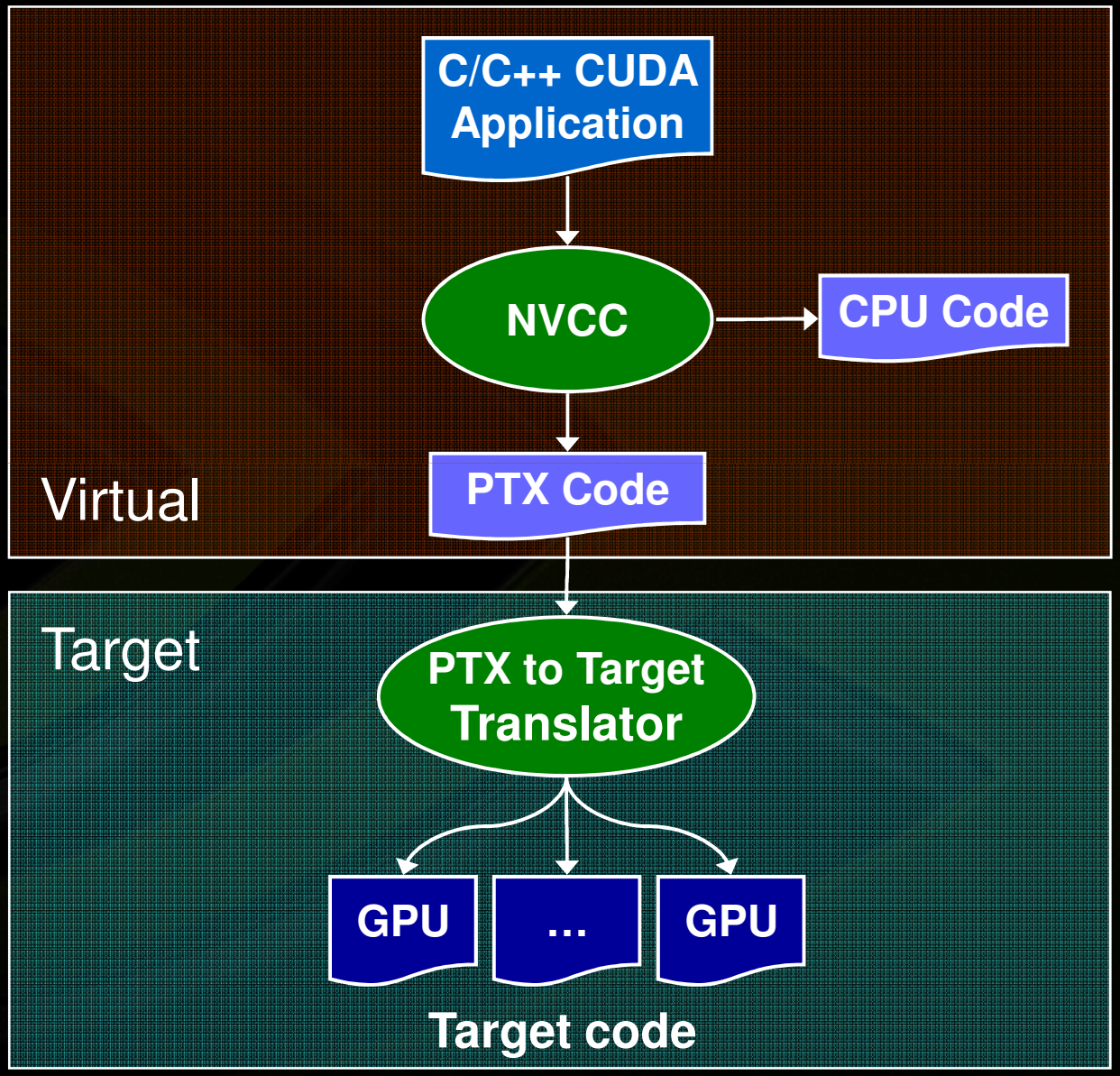
Autotuning

- Both extremes seem to be too good to be true
- We'll probably end up in the middle
  - Programmer will do some of the parameterization
    - Identify blocks
    - Memory tradeoffs
    - Serial code
  - Autotuners explores smaller space

Composition is key

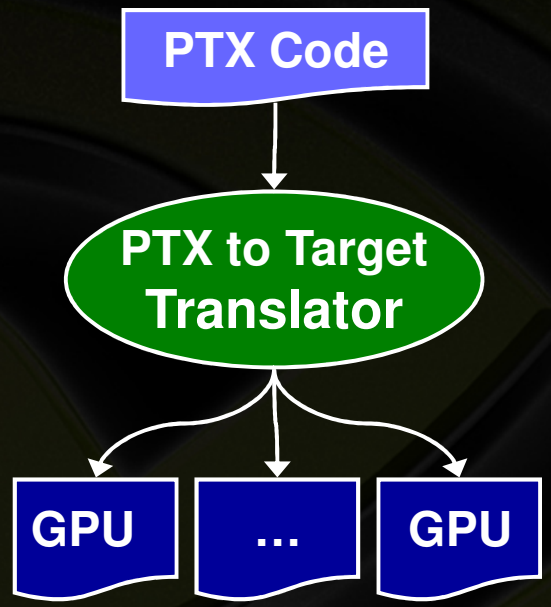
- Tuners likely need access to complete code base
- Need powerful/expressive enough IL that isnt source
  - Allow investment
- Client side must be smart upfront, or binary ships its own brains
  - Smart client
    - can have IL logic for local system, supplied perhaps by IHVs
  - Smart binary
    - more flexible but may not understand the target
- Seems desirable for IL to include high level expression

Compiling CUDA



Virtual to Target ISA Translation

```
ld.global.v4.f32  {$f1,$f3,$f5,$f7},[$r9+0];
mad.f32           $f1,$f5,$f3,$f1;
```



Target code

```
0x103c8009 0x0fffffff
0xd00e0609 0xa0c00780
0x100c8009 0x00000003
0x21000409 0x07800780
```

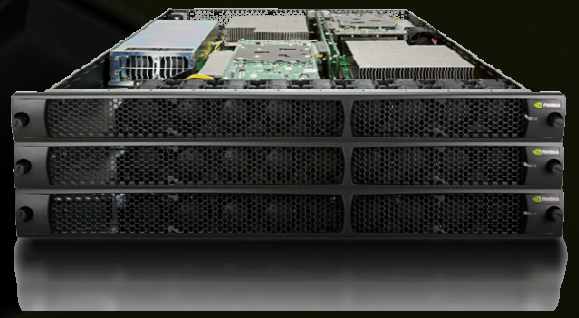
- Parallel Thread eXecution (PTX)
  - Virtual Machine and ISA
  - Distribution format for applications
  - Install-time translation
  - “fat binary” caches target-specific versions
- Target-specific translation optimizes for:
  - ISA diffences
  - Resource allocation
  - Performance

Interesting Architectures

- Do more on GPUs
  - Millions out there
  - Compact, well suited for server farms
  - Plenty of tuning parameters
  - A very hard problem
  - Represents many issues many-core CPUs are going to
  - Its like the future – Today

Interesting Architectures

- Heterogeneous Tuning
  - Figuring out how to divide work appropriately among asymmetrical cores
  - E.g. partitioning a problem to map serial code onto an aggressive out-of-order mono-core CPU plus parallel parts of problem onto a plenty core GPU.



Questions?

Ben de Waal
ben@nvidia.com