NVIDIA

N-WAYS GPU BOOTCAMP

OUT OF SCOPE

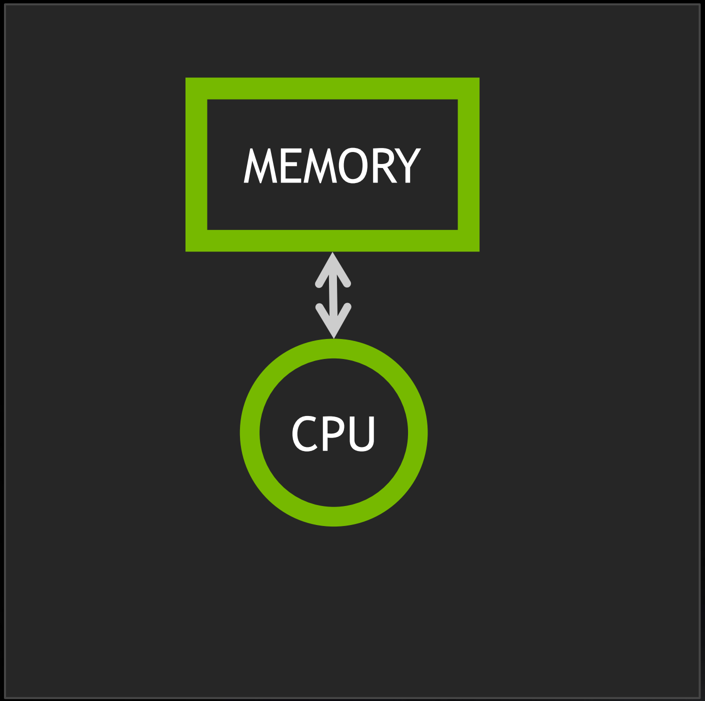
- This session should not be considered as extensive guide covering all the API of respective parallel programming model
- This session is focused on the introduction and no detail optimization will covered

INTRODUCTION TO GPU COMPUTING

What to expect?

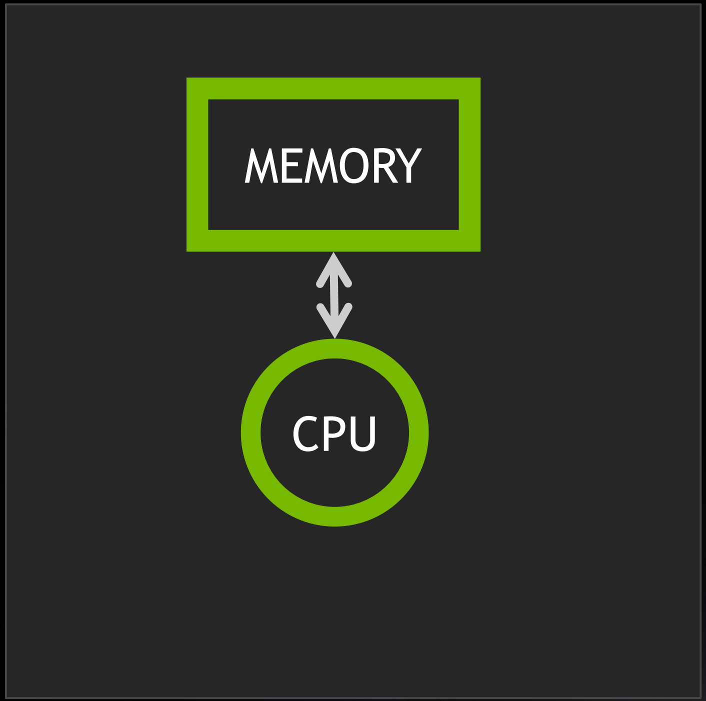
- Broad view on GPU Stack
- Fundamentals of GPU Architecture
- Good starting point

HPC SYSTEM EVOLUTION

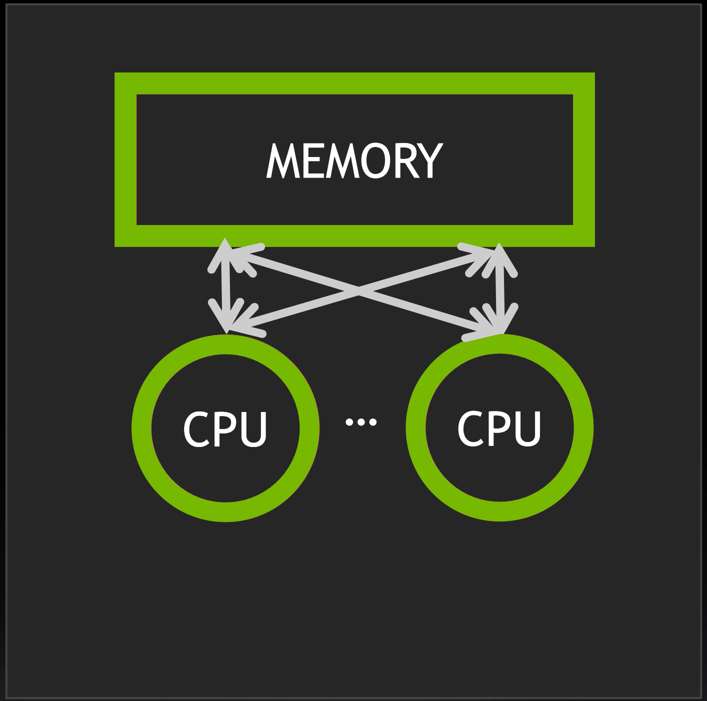


Sequential

HPC SYSTEM EVOLUTION

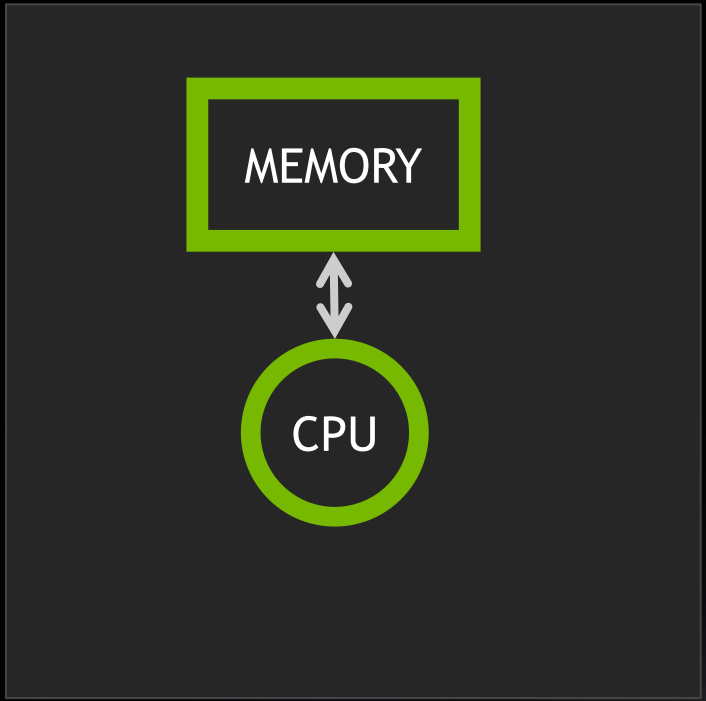


Sequential

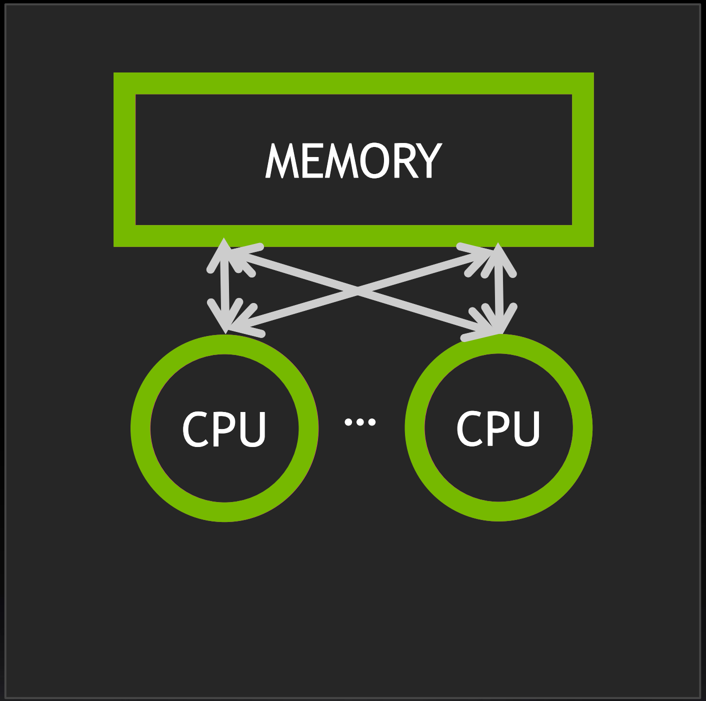


Multithreaded
P-Thread/OpenMP

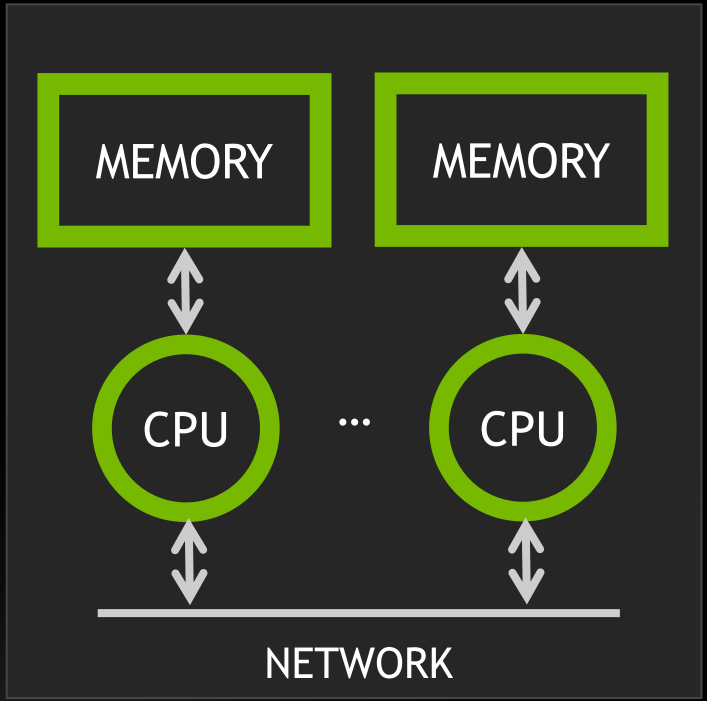
HPC SYSTEM EVOLUTION



Sequential

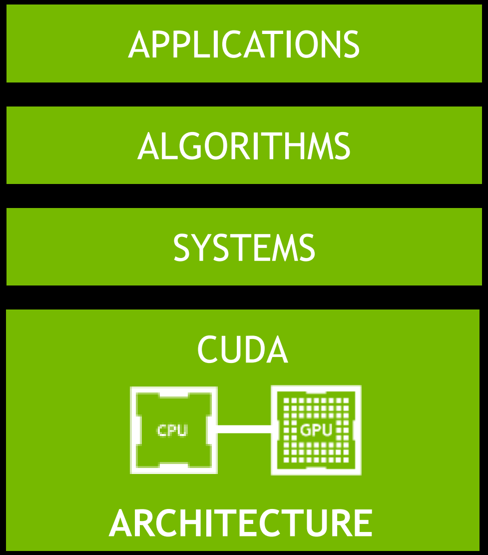


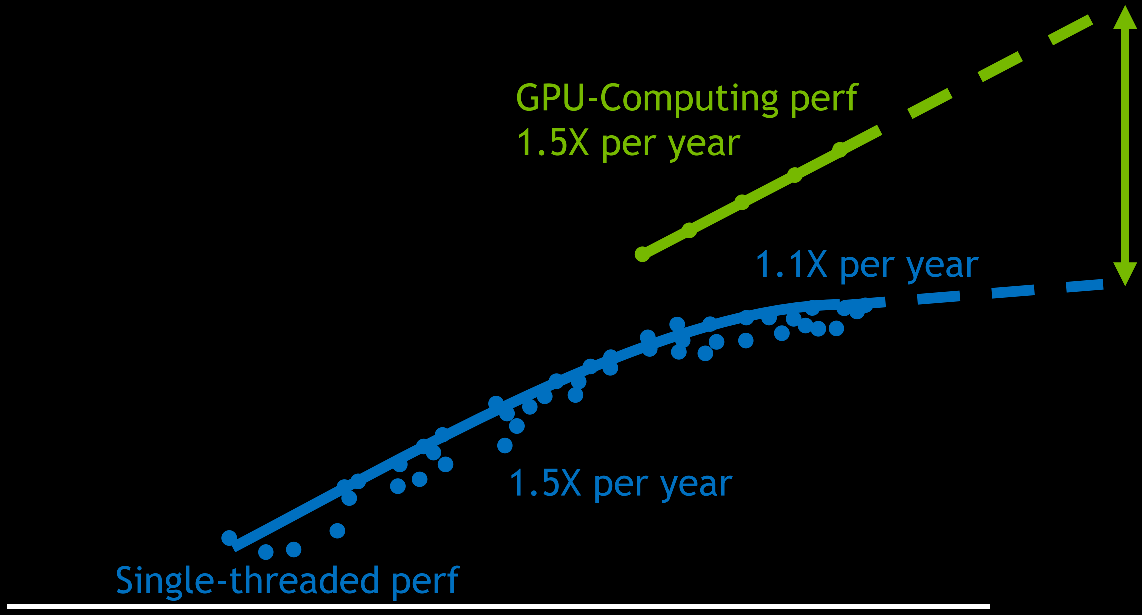
Multithreaded
P-Thread/OpenMP



Distributed
MPI

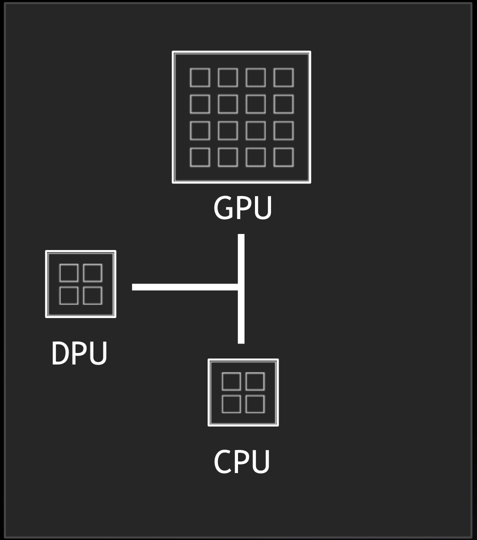
GPU ARCHITECTURE CONTINUES TO DELIVER PERFORMANCE



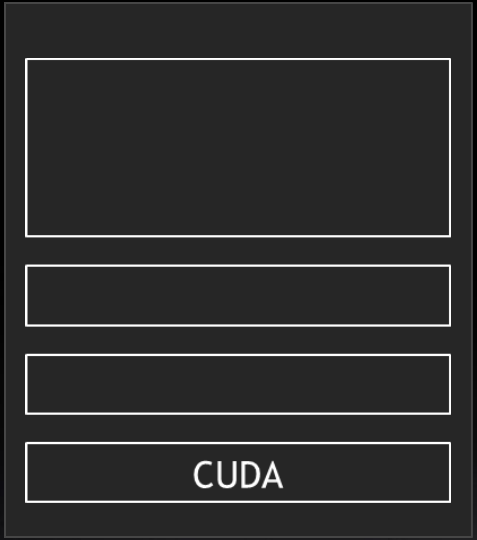


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten New plot and data collected for 2010-2015 by K. Rupp

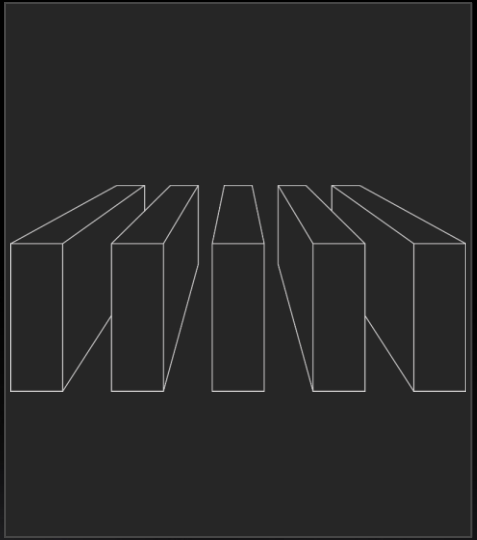
ACCELERATED COMPUTING PILLARS



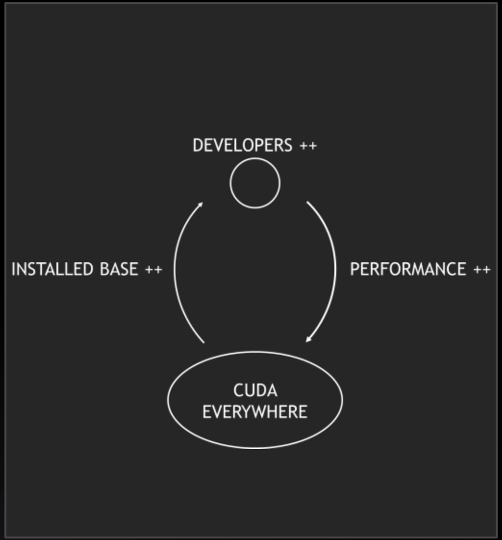
X-FACTOR SPEED UP



FULL STACK

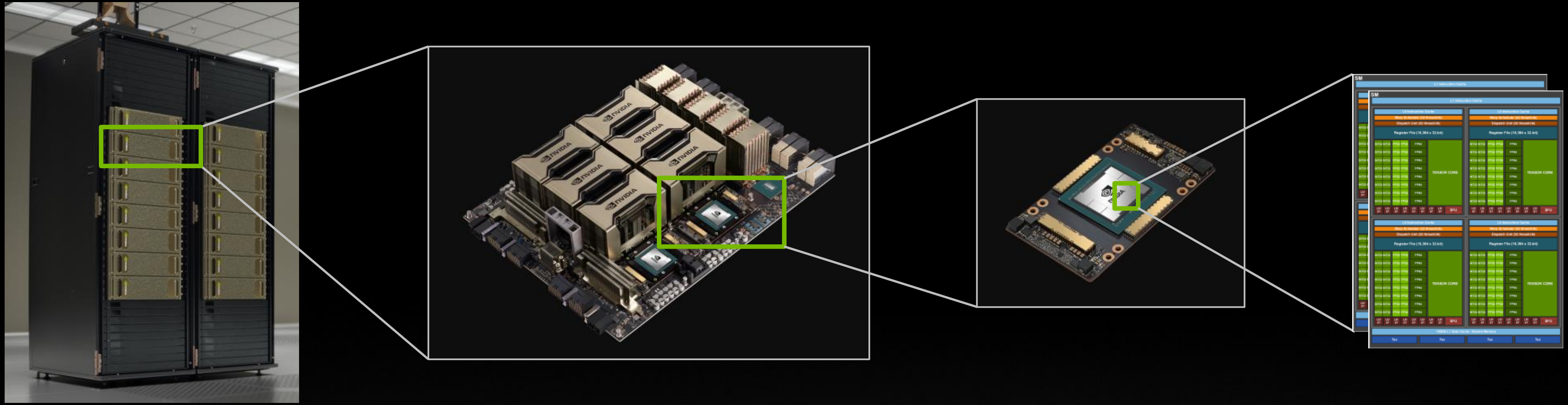


DATA-CENTER SCALE



Developer Productivity

HIERARCHY OF SCALES



Multi-System Rack
Unlimited Scale

Multi-GPU System
8 GPUs

Multi-SM GPU
108 Multiprocessors

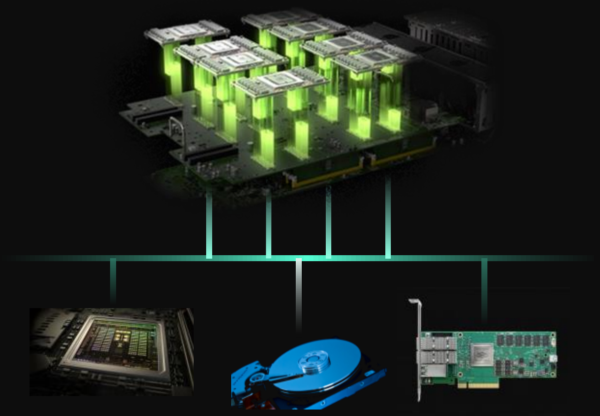
Multi-Core SM
2048 threads

CUDA PLATFORM: TARGETS EACH LEVEL OF THE HIERARCHY

The CUDA Platform Advances State Of The Art From Data Center To The GPU

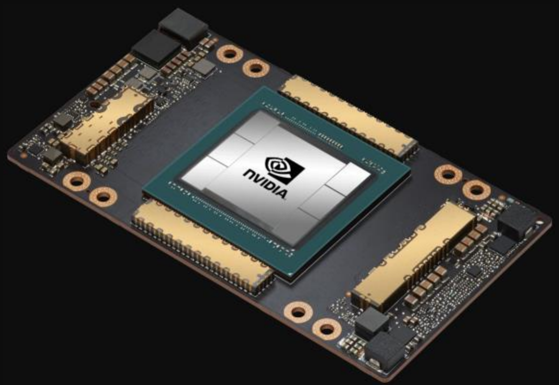
System Scope

FABRIC MANAGEMENT
DATA CENTER OPERATIONS
DEPLOYMENT
MONITORING
COMPATIBILITY
SECURITY



Node Scope

GPU-DIRECT
NVLINK
LIBRARIES
UNIFIED MEMORY
ARM
MIG

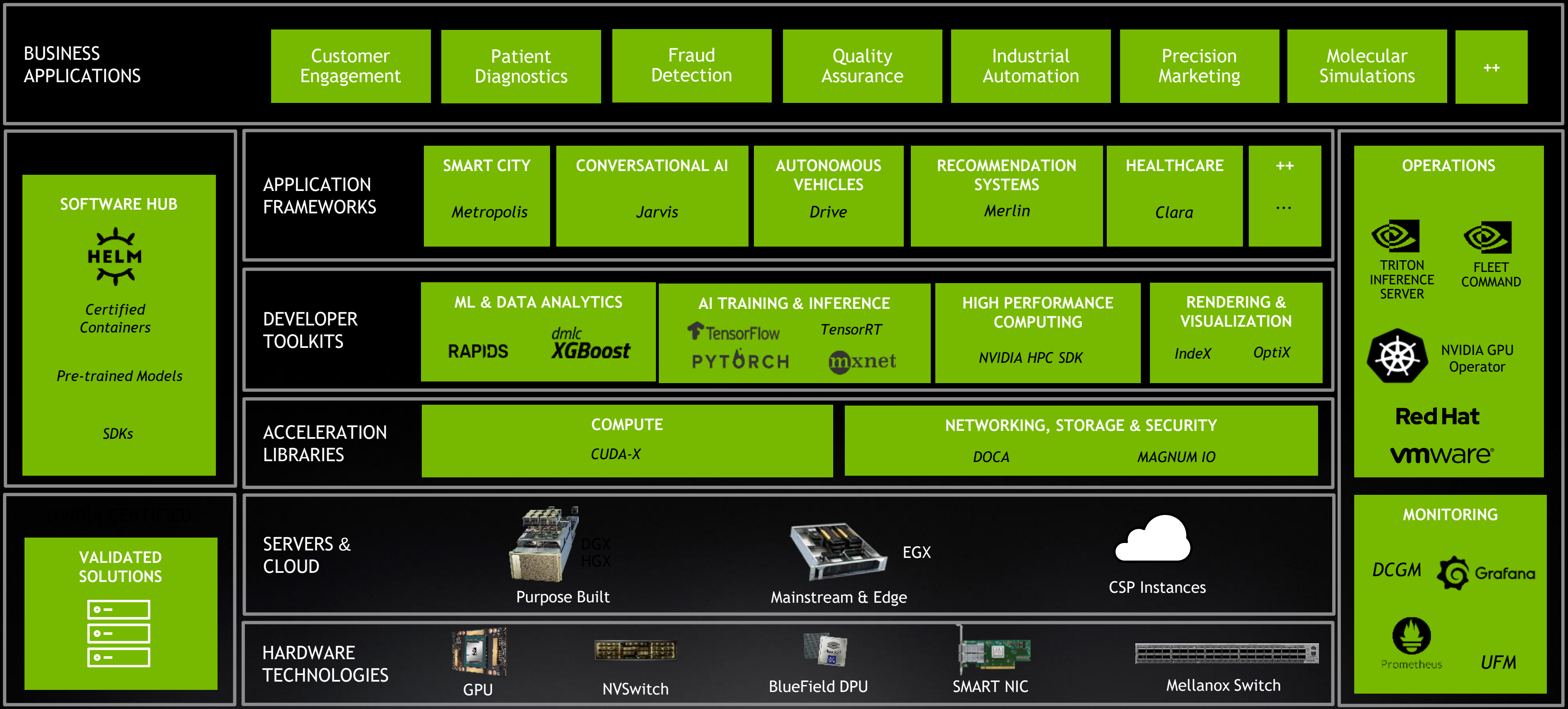


Program Scope

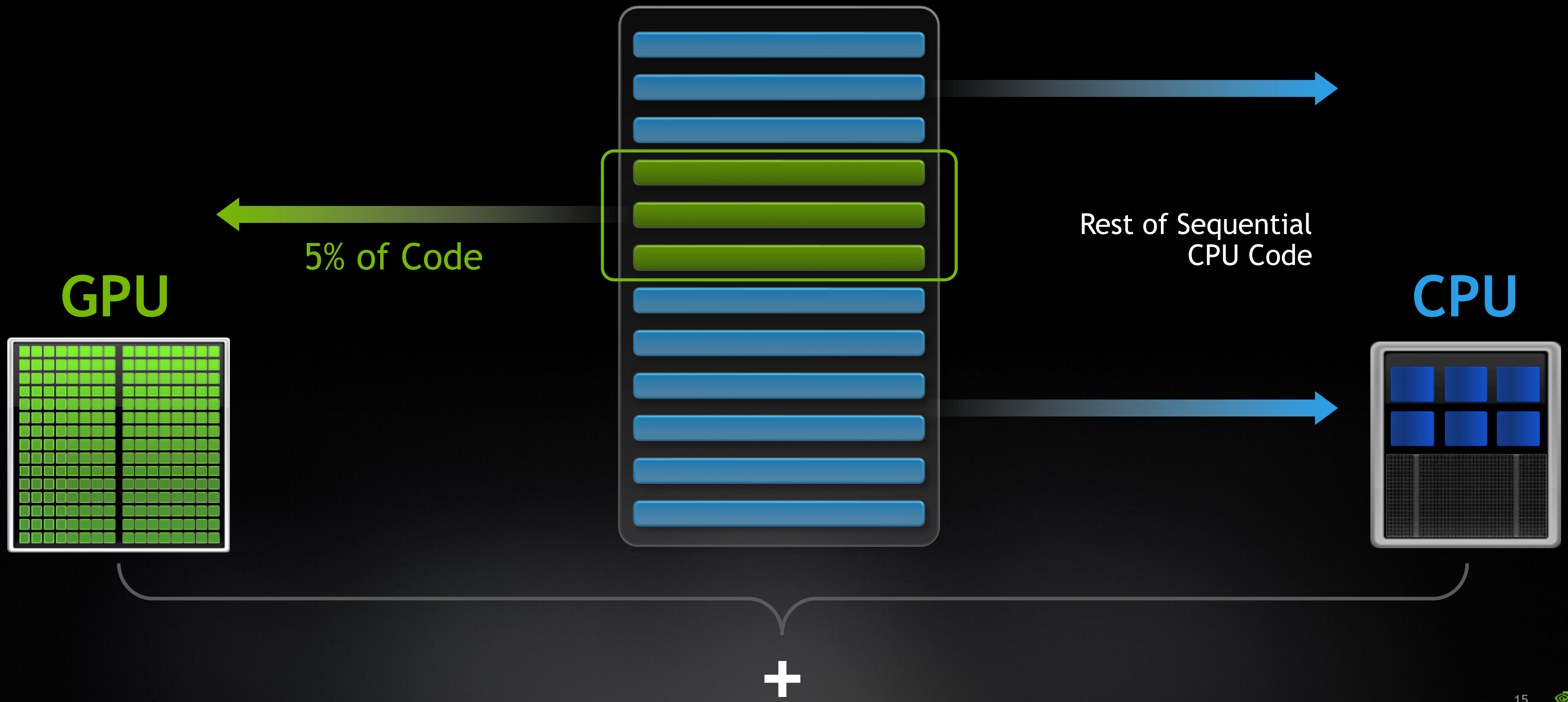
CUDA C++
OPENACC
STANDARD LANGUAGES
SYNCHRONIZATION
PRECISION

SCOPE OF THIS SESSION

ACCELERATED PLATFORM



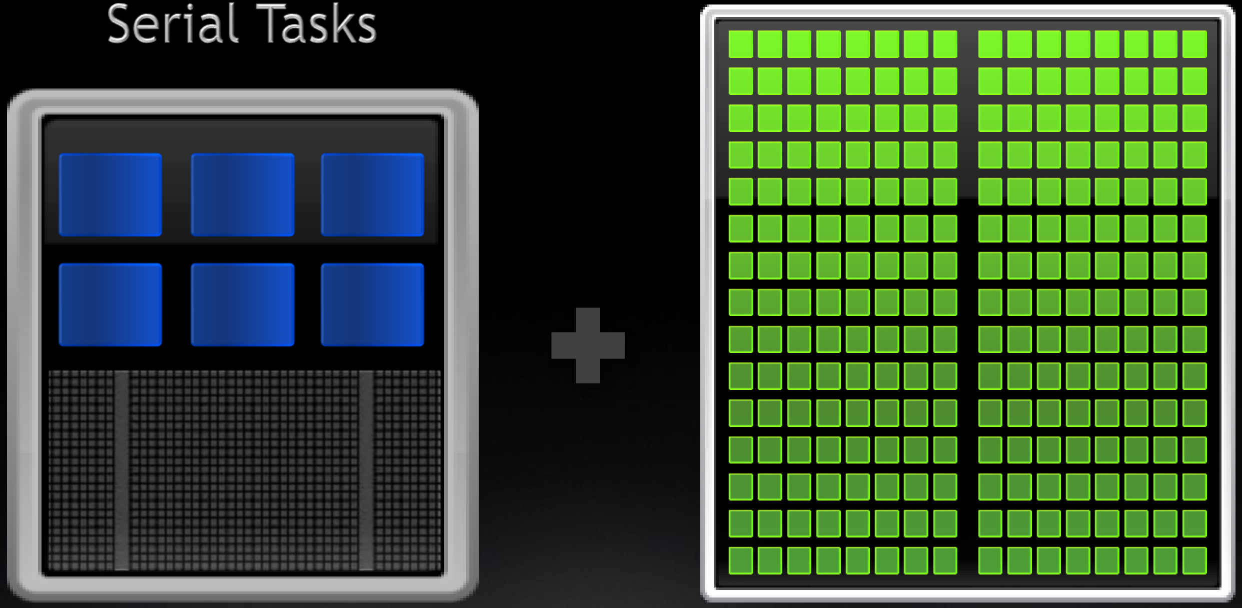
HOW GPU ACCELERATION WORKS



ACCELERATED COMPUTING

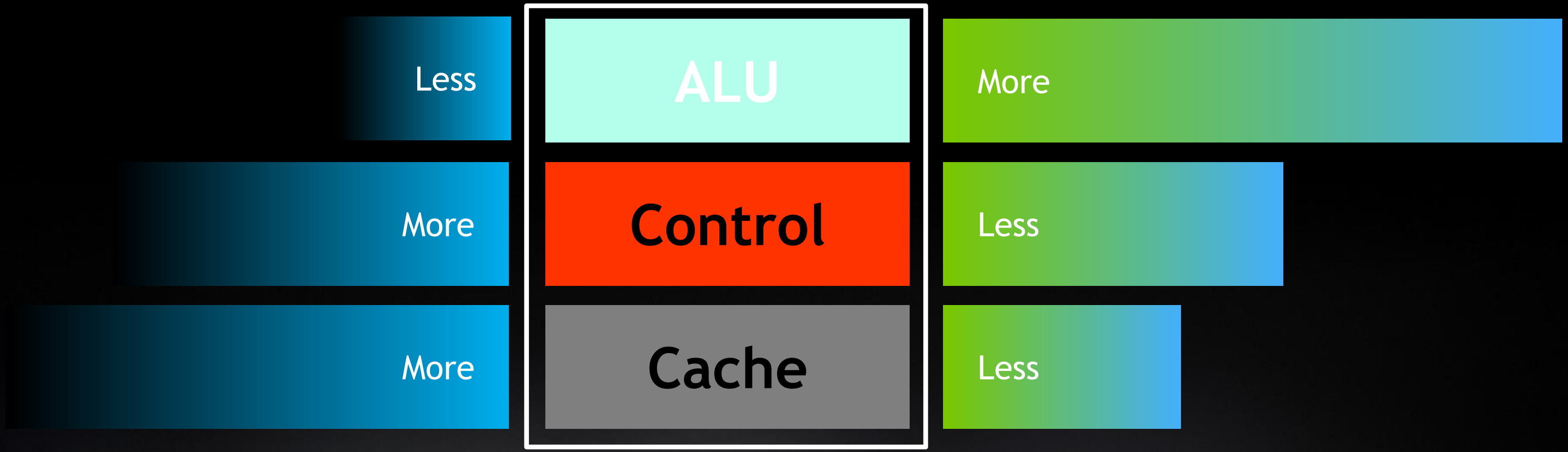
CPU
Optimized for
Serial Tasks

GPU Accelerator
Optimized for
Parallel Tasks



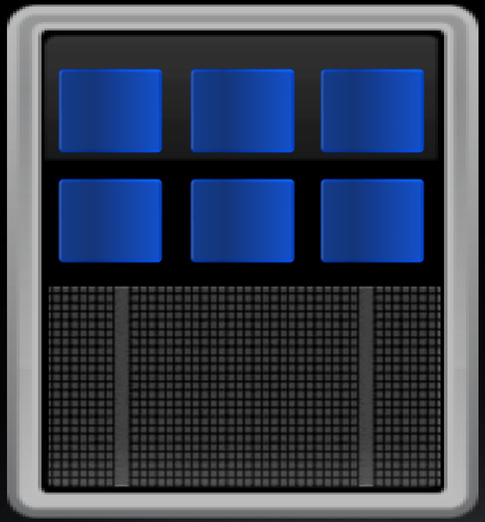
SILICON BUDGET

- The three components of any processor



CPU IS A LATENCY REDUCING ARCHITECTURE

CPU
Optimized for
Serial Tasks



CPU Strengths

- Very large main memory
- Very fast clock speeds
- Latency optimized via large caches
- Small number of threads can run very quickly

CPU Weaknesses

- Relatively low memory bandwidth
- Cache misses very costly
- Low performance/watt

GPU IS ALL ABOUT HIDING LATENCY

GPU Strengths

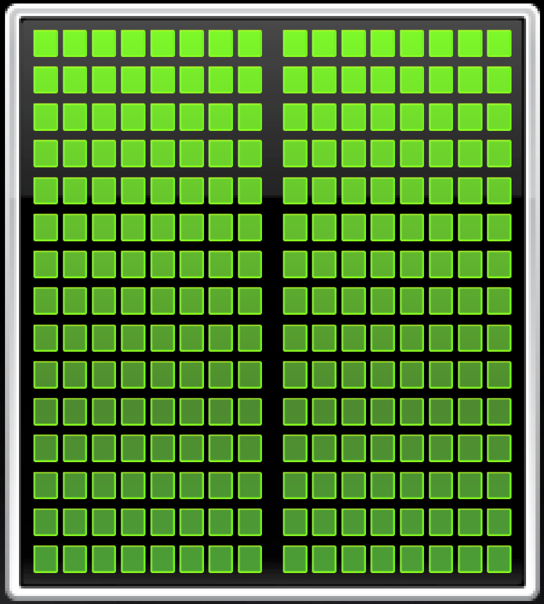
- High bandwidth main memory
- Significantly more compute resources
- Latency tolerant via parallelism
- High throughput
- High performance/watt

GPU Weaknesses

- Relatively low memory capacity
- Low per-thread performance

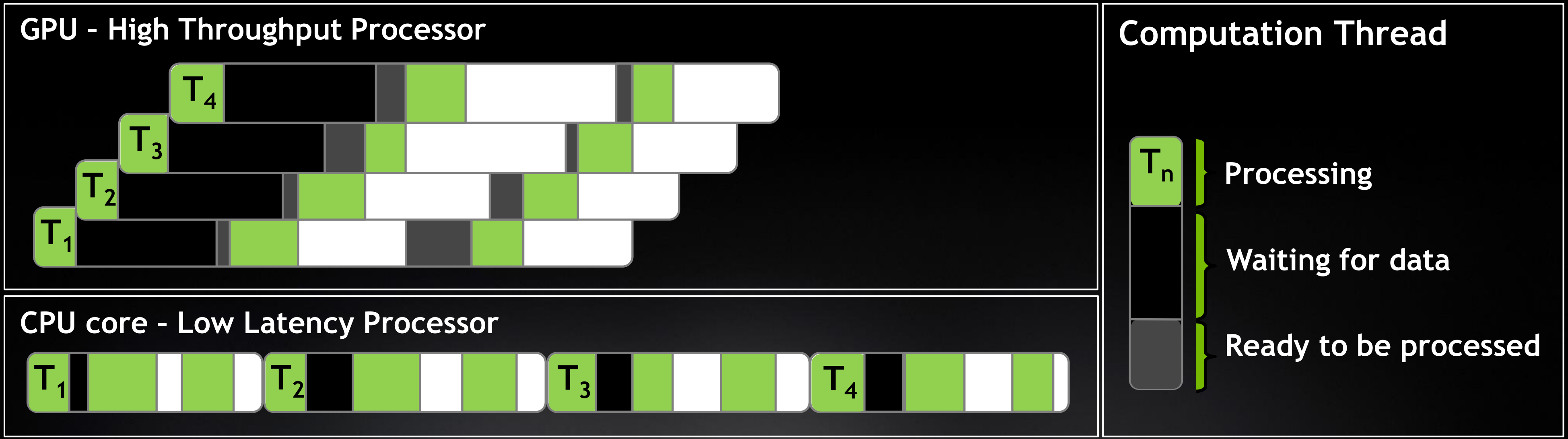
GPU Accelerator

Optimized for
Parallel Tasks



LOW LATENCY VS HIGH THROUGHPUT

- CPU architecture must minimize latency within each thread
- GPU architecture hides latency with computation (data-parallelism, to 30k threads!)



SPEED V. THROUGHPUT

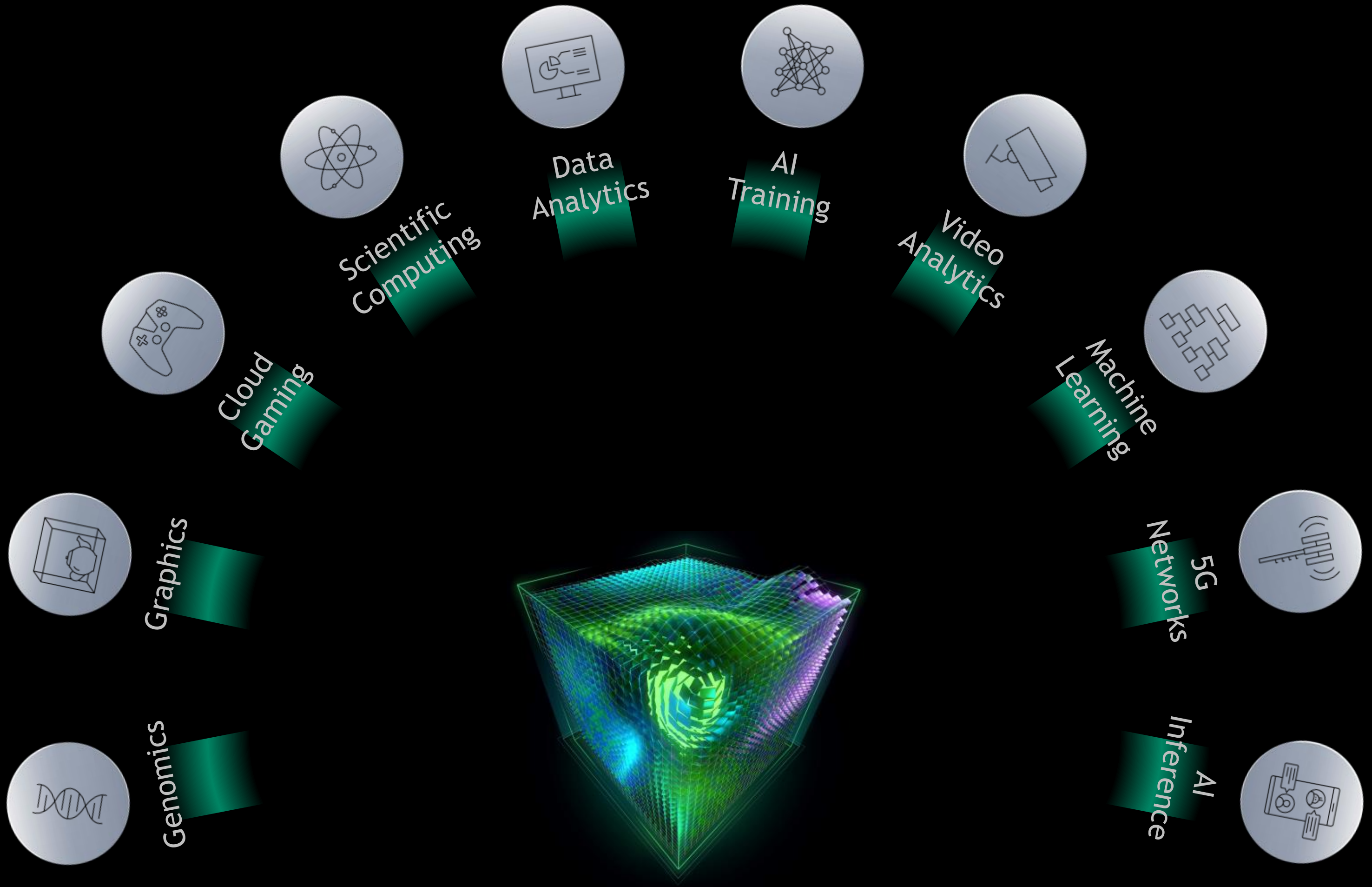
Speed

Throughput

Which is better depends on your needs…

*Images from Wikimedia Commons via Creative Commons

HUGE BREADTH OF PLATFORMS, SYSTEMS, LANGUAGES



NVIDIA HPC SDK

Download at developer.nvidia.com/hpc-sdk

NVIDIA HPC SDK

DEVELOPMENT

Compilers
- nvcc
- nvc
- nvc++
- nvfortran

Math Libraries
- cuBLAS
- cuTENSOR
- cuSPARSE
- cuSOLVER
- cuFFT
- cuRAND

Communication Libraries
- Open MPI
- NVSHMEM
- NCCL

Programming Models
- Standard C++ & Fortran
- OpenACC & OpenMP
- CUDA

ANALYSIS

Profilers
- Nsight
  - Systems
  - Compute

Debugger
- cuda-gdb
  - Host
  - Device

DEPLOYMENT

Container
- HPC Container Maker / NVIDIA Container Runtime

Develop for the NVIDIA HPC Platform: GPU, CPU and Interconnect

HPC Libraries | GPU Accelerated C++ and Fortran | Directives | CUDA

N-WAYS TO GPU PROGRAMMING

Math Libraries | Standard Languages | Directives | CUDA

```
std::transform(par, x, x+n, y, y,
    [=](float x, float y) {
        return y + a*x;
});
```

```
do concurrent (i = 1:n)
   y(i) = y(i) + a*x(i)
enddo
```

GPU Accelerated C++ and Fortran

```
#pragma acc data copy(x,y)
{

...

std::transform(par, x, x+n, y, y,
    [=](float x, float y) {
        return y + a*x;
});

...

}
```

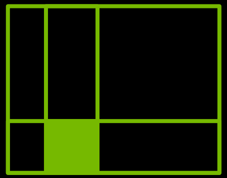
Incremental Performance Optimization with Directives

```
__global__
void saxpy(int n, float a,
           float *x, float *y) {
  int i = blockIdx.x*blockDim.x +
          threadIdx.x;
  if (i < n) y[i] += a*x[i];
}

int main(void) {
  cudaMallocManaged(&x, ...);
  cudaMallocManaged(&y, ...);
  ...
  saxpy<<<(N+255)/256,256>>>(...,x, y)
  cudaDeviceSynchronize();
  ...
}
```

Maximize GPU Performance with CUDA C++/Fortran

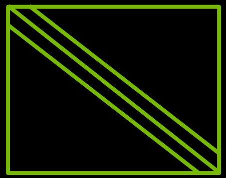
GPU Accelerated Math Libraries

GPU ACCELERATED MATH LIBRARIES



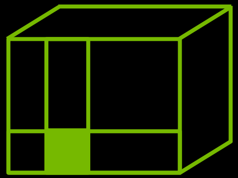
cuBLAS

BF16, TF32 and FP64 Tensor Cores



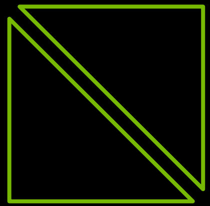
cuSPARSE

Increased memory BW, Shared Memory & L2



cuTENSOR

BF16, TF32 and FP64 Tensor Cores

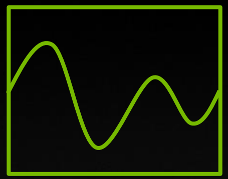


cuSOLVER

BF16, TF32 and FP64 Tensor Cores

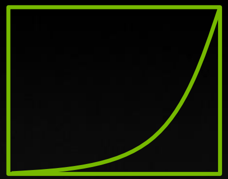
nvJPEG

Hardware Decoder



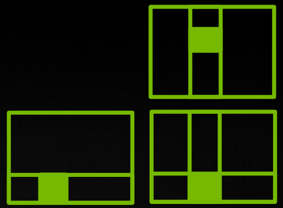
cuFFT

BF16, TF32 and FP64 Tensor Cores



CUDA Math API

Increased memory BW, Shared Memory & L2

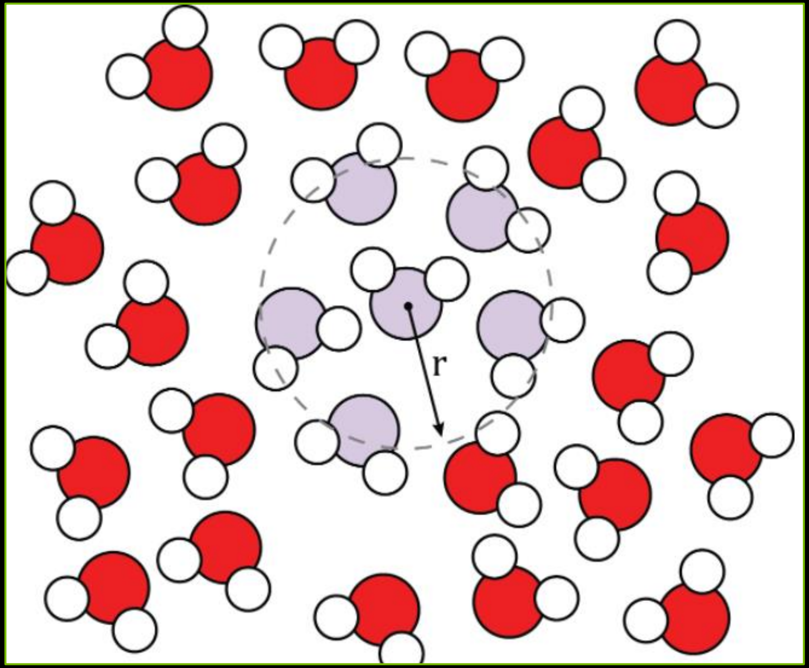


CUTLASS

BF16 & TF32 Support

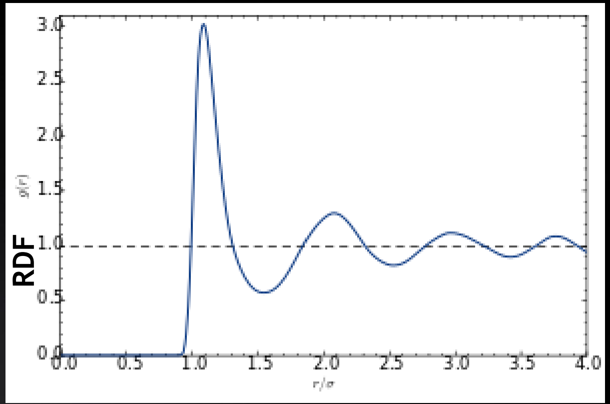
APPLICATION

Molecular Simulation



RDF

The radial distribution function (RDF) denoted in equations by g(r) defines the probability of finding a particle at a distance r from another tagged particle.



https://en.wikipedia.org/wiki/Radial_distribution_function

RDF

Pseudo Code - C

```
for (int frame=0;frame<nconf;frame++){

    for(int id1=0;id1<numatm;id1++)
    {
        for(int id2=0;id2<numatm;id2++)
        {
            dx=d_x[]-d_x[];
            dy=d_y[]-d_y[];
            dz=d_z[]-d_z[];
            r=sqrtf(dx*dx+dy*dy+dz*dz);

            if (r<cut) {
                ig2=(int)(r/del);
                d_g2[ig2] = d_g2[ig2] +1 ;
            }
        }
    }
}
```

- Across Frames
- Find Distance
- Reduction

RDF

Pseudo Code - Fortran

```
do iconf=1,nframes
   if (mod(iconf,1).eq.0) print*,iconf

   do i=1,natoms
       do j=1,natoms
                dx=x(iconf,i)-x(iconf,j)
                dy=y(iconf,i)-y(iconf,j)
                dz=z(iconf,i)-z(iconf,j)

                r=dsqrt(dx**2+dy**2+dz**2)
                if(r<cut)then
                        g(ind)=g(ind)+1.0d0
                endif
        enddo
   enddo
enddo
```

- Across Frames
- Find Distance
- Reduction

SINGLE PRECISION ALPHA X PLUS Y (SAXPY)

GPU SAXPY in multiple languages and libraries

Part of Basic Linear Algebra Subroutines (BLAS) Library

$$z = \alpha x + y$$

x, y, z : vector

α : scalar

SAXPY: OPENACC COMPILER DIRECTIVES

Parallel C Code

```c
void saxpy(int n,
           float a,
           float *x,
           float *y)
{
#pragma acc kernels
  for (int i = 0; i <
    y[i] = a*x[i] + y[i];
}

...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```

Parallel Fortran Code

```fortran
subroutine saxpy(n, a, x, y)
  real :: x(:), y(:), a
  integer :: n, i
!$acc kernels
  do i=1,n
    y(i) = a*x(i)+y(i)
  enddo
!$acc end kernels
end subroutine saxpy


...
! Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...
```

www.openacc.org

SAXPY: CUBLAS LIBRARY

Serial BLAS Code

```
int N = 1<<20;

...

// Use your choice of blas library

// Perform SAXPY on 1M elements
blas_saxpy(N, 2.0, x, 1, y, 1);
```

Parallel cuBLAS Code

```
int N = 1<<20;

cublasInit();
cublasSetVector(N, sizeof(x[0]), x, 1, d_x, 1);
cublasSetVector(N, sizeof(y[0]), y, 1, d_y, 1);

// Perform SAXPY on 1M elements
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);

cublasGetVector(N, sizeof(y[0]), d_y, 1, y, 1);

cublasShutdown();
```

You can also call cuBLAS from Fortran, C++, Python, and other languages:
http://developer.nvidia.com/cublas

SAXPY: CUDA C

Standard C

```
void saxpy(int n, float a,
           float *x, float *y)
{
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

int N = 1<<20;



// Perform SAXPY on 1M elements
saxpy(N, 2.0, x, y);
```

Parallel C

```
__global__
void saxpy(int n, float a,
           float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx
          ) y[i] = a*x[i] + y[i];
}

int N = 1<<20;
cudaMemcpy(d_x, x, N, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N, cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<4096,256>>>(N, 2.0, d_x, d_y);

cudaMemcpy(y, d_y, N, cudaMemcpyDeviceToHost);
```

http://developer.nvidia.com/cuda-toolkit

SAXPY: CUDA FORTRAN

Standard Fortran

```
module mymodule contains
  subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, i
    do i=1,n
      y(i) = a*x(i)+y(i)
    enddo
  end subroutine saxpy
end module mymodule

program main
  use mymodule
  real :: x(2**20), y(2**20)
  x = 1.0, y = 2.0

  ! Perform SAXPY on 1M elements
  call saxpy(2**20, 2.0, x, y)

end program main
```

Parallel Fortran

```
module mymodule contains
  attributes(global) subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, i
    attributes(value) :: a, n
    i = threadIdx%x+(blockIdx%x-1)*blockDim%x
    if (i<=n) y(i) = a*x(i)+y(i)
  end subroutine saxpy
end module mymodule

program main
  use cudafor; use mymodule
  real, device :: x_d(2**20), y_d(2**20)
  x_d = 1.0, y_d = 2.0

  ! Perform SAXPY on 1M elements
  call saxpy<<<4096,256>>>(2**20, 2.0, x_d, y_d)

end program main
```

http://developer.nvidia.com/cuda-fortran

SAXPY: PYTHON

Standard Python

```
import numpy as np


def saxpy(a, x, y):
  return [a * xi + yi
          for xi, yi in zip(x, y)]

x = np.arange(2**20, dtype=np.float32)
y = np.arange(2**20, dtype=np.float32)




cpu_result = saxpy(2.0, x, y)
```

http://numpy.scipy.org

Numba: Parallel Python

```
import numpy as np
from numba import vectorize

@vectorize(['float32(float32, float32,
float32)'], target='cuda')
def saxpy(a, x, y):
    return a * x + y

N = 1048576

# Initialize arrays
A = np.ones(N, dtype=np.float32)
B = np.ones(A.shape, dtype=A.dtype)
C = np.empty_like(A, dtype=A.dtype)

# Add arrays onGPU
C = saxpy(2.0, X, Y)
```

https://numba.pydata.org

SAXPY: PYTHON

Standard Python

```python
import numpy as np


x = np.ones(2**20, dtype=np.float32)
y = np.ones(2**20, dtype=np.float32)

def saxpy(a, x, y):
  return a * x + y

cpu_result = saxpy(2.0, x, y )
```

http://numpy.scipy.org

CUPY: GPU accelerated NumPy Python

```python
import cupy as cp

# Initialize arrays
x = cp.ones(2**20, dtype=cp.float32)
y = cp.ones(2**20, dtype=cp.float32)

def saxpy(a, x, y):
  return a * x + y

gpu_result = saxpy(2.0, x, y)
```

https://github.com/cupy/cupy

SAXPY: MATLAB

Parallel C Code

```
void saxpy(int n,
           float a,
           float *x,
           float *y)
{
#pragma acc kernels
  for (int i = 0; i <
    y[i] = a*x[i] + y[i];
}

...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```

Parallel C++ Code

```
<<initialize>>
p = parpool

parfor i = 1:numel(N)
   y(i) = 2.0 * x(i) + y(i)
end

<<post process>
delete(p)
```

SAXPY: MATLAB

- 500+ GPU-enabled MATLAB functions
- Additional GPU-enabled Toolboxes
  - Neural Networks
  - Image Processing and Computer Vision
  - Communications
  - Signal Processing
  - Stats Toolbox

Transfer Data To GPU From

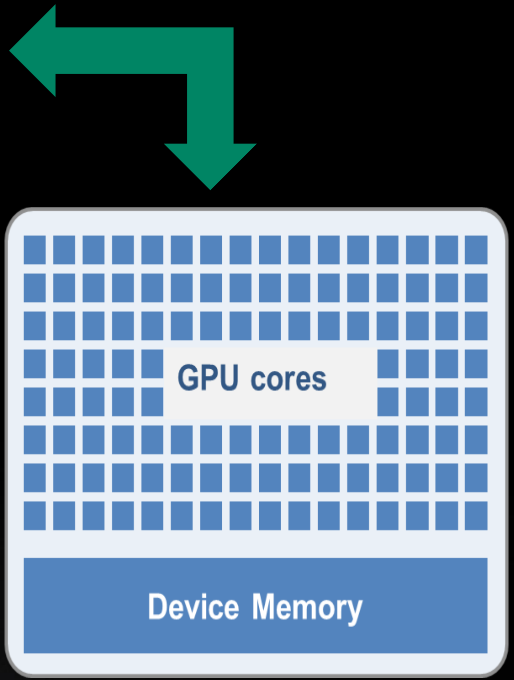
```
x=gpuArray(x);
```

Perform Calculation on GPU

```
X=saxpy(N,2.0,x,0,1,y,0,1);
```

Gather Data or Plot

```
y=gather(y)
```



[MATLAB GPU computing]

SAXPY: PSTL

Serial C++ Code (with STL and Boost)

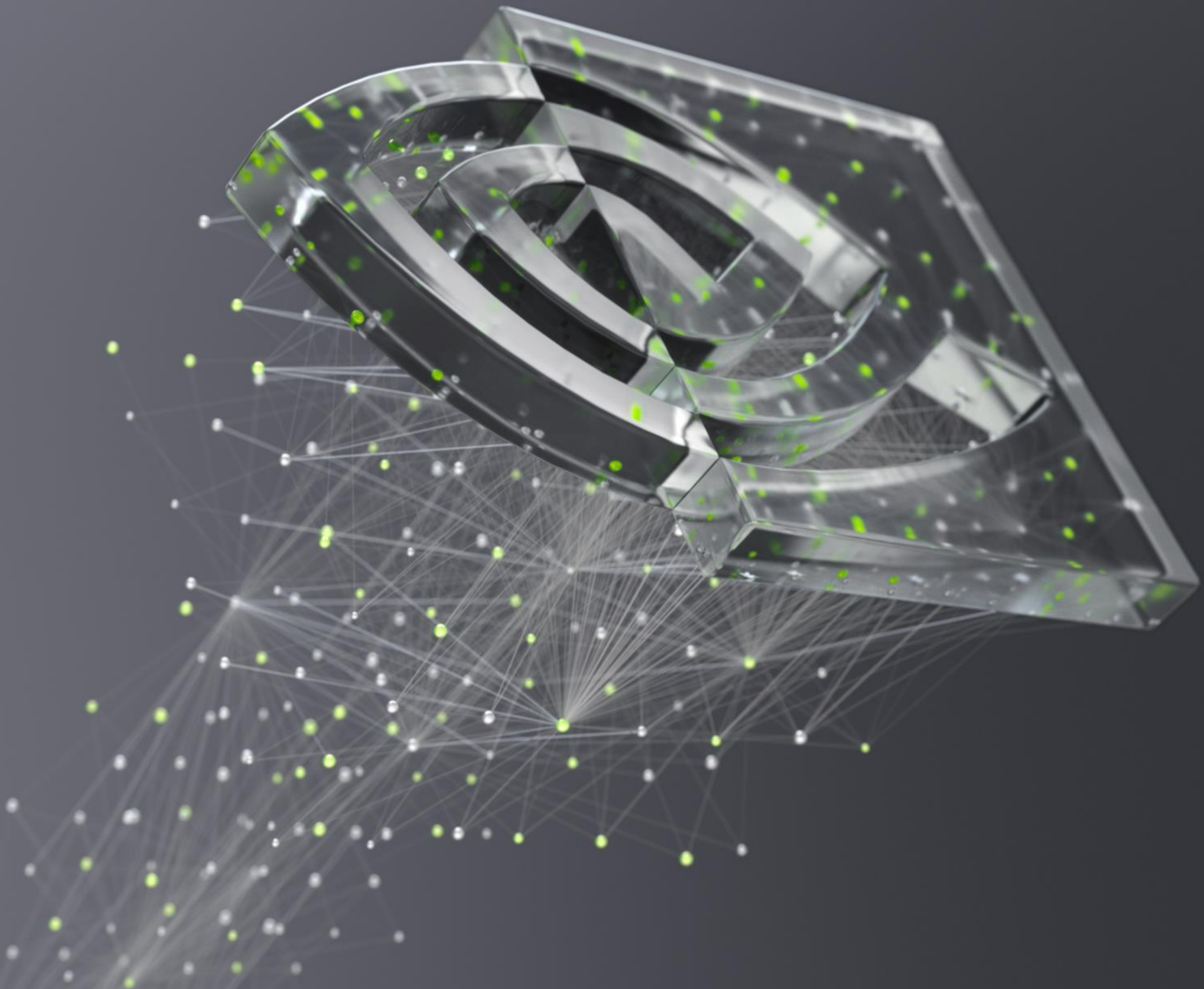
```
int N = 1<<20;
std::vector<float> x(N), y(N);

...




// Perform SAXPY on 1M elements
std::transform(x.begin(), x.end(),
               y.begin(), y.end(),
          2.0f * _1 + _2);
```

Parallel C++ Code

```
int N = 1<<20;
std::vector<float> x(N), y(N);

...




// Perform SAXPY on 1M elements
std::transform(std::execution::par
, x.begin(), x.end(),
               y.begin(), y.end(),
          2.0f * _1 + _2);
```

www.boost.org/libs/lambda

THANK YOU



BACKUP