Agenda

1. Introduction: Accelerated processing & GPUs
3. Tesla Platform for HPC
4. CPU architecture & GPU architecture
5. Tesla Platform for Developers
6. Embedded Processing: Jetson TX1
RACING TOWARD EXASCALE

Exascale System Sketch

25 Years Ago
Vector Systems

15 Years Ago
Distributed Systems

Today
Heterogeneous Systems

Future

The Two Primary Challenges to Overcome

Energy Efficiency
23x in 7 years

Parallel (10^4 threads)
Hierarchical
Heterogeneous

Programming

In the Future
All performance is from processors
Systems are power limited
Efficiency & performance
Systems are communication limited
(Frequency is performance)
THE WORLD LEADER IN VISUAL COMPUTING
<table>
<thead>
<tr>
<th>SIMULATION</th>
<th>MACHINE LEARNING</th>
<th>VISUALIZATION</th>
</tr>
</thead>
</table>

**TESLA ACCELERATED COMPUTING**
Accelerated Computing Roadmap
PERFORMANCE LEAD CONTINUES TO GROW

**Peak Double Precision FLOPS**

- NVIDIA GPU: Green line
- x86 CPU: Blue line

**Peak Memory Bandwidth**

- NVIDIA GPU: Green line
- x86 CPU: Blue line
GPU Architecture Roadmap

- **Tesla** (2008)
- **Fermi** (2010)
- **Kepler** (2012)
- **Maxwell** (2014)
- **Pascal** (2016)

Features:
- Mixed Precision
- 3D Memory
- NVLink
TESLA ACCELERATED COMPUTING PLATFORM

Focused on Co-Design from Top to Bottom

Fast GPU
Engineered for High Throughput

Productive Programming Model & Tools

Expert Co-Design

Accessibility

TFLOPS

NVIDIA GPU

x86 CPU

2008 2009 2010 2011 2012 2013 2014

TFLOPS

0.0

0.5

1.0

1.5

2.0

2.5

3.0

M1060

M2090

K20

K40

K80

M2090

K20

K40

K80

Fast GPU

Strong CPU

APPLICATION

MIDDLEWARE

SYS SW

LARGE SYSTEMS

PROCESSOR
# TESLA GPU ACCELERATORS 2015-2016*

<table>
<thead>
<tr>
<th>2015</th>
<th>2016</th>
<th>2017</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>KEPLER - K80</strong></td>
<td><strong>MAXWELL - M40</strong></td>
<td><strong>GRID Enabled</strong></td>
</tr>
<tr>
<td>2xGPU, 2.9TF DP, 8.7TF SP Peak</td>
<td>1xGPU, 7TF SP Peak (Boost Clock), 12GB, 288 GB/s, 250W PCIe Passive</td>
<td><strong>Enabled</strong></td>
</tr>
<tr>
<td>4.4TF SGEMM/1.59TF DGEMM 24GB, ~480 GB/s, 300W PCIe Passive</td>
<td><strong>GRID Enabled</strong></td>
<td></td>
</tr>
<tr>
<td><strong>KEPLER - K40</strong></td>
<td><strong>MAXWELL - M40</strong></td>
<td><strong>MAXWELL - M4</strong></td>
</tr>
<tr>
<td>1.43TF DP, 4.3TF SP Peak 3.3 TF SGEMM/1.22TF DGEMM 12 GB, 288 GB/s, 235W PCIe Active/PCIe Passive</td>
<td><strong>GRID Enabled</strong></td>
<td>1xGPU, 2.2 TF SP Peak, 4GB, 88 GB/s, 50-75W, PCIe Low Profile</td>
</tr>
<tr>
<td><strong>MAXWELL - M40</strong></td>
<td><strong>MAXWELL - M40</strong></td>
<td><strong>MAXWELL - M40</strong></td>
</tr>
<tr>
<td>2xGPU, 7.4TF SP Peak, -6TF SGEMM 16GB, 320 GB/s, 300W PCIe Active/PCIe Passive</td>
<td>1xGPU, TBD TF SP Peak, 12GB, 288 GB/s, 250W PCIe Passive</td>
<td><strong>GRID Enabled</strong></td>
</tr>
<tr>
<td><strong>MAXWELL - M40</strong></td>
<td><strong>MAXWELL - M40</strong></td>
<td><strong>MAXWELL - M40</strong></td>
</tr>
<tr>
<td>1xGPU, TBD TF SP Peak, 8GB, 160 GB/s, 75-100W, MXM</td>
<td><strong>GRID Enabled</strong></td>
<td><strong>Enabled</strong></td>
</tr>
</tbody>
</table>

*For End Customer Deployments

NVIDIA CONFIDENTIAL. DO NOT DISTRIBUTE.
<table>
<thead>
<tr>
<th></th>
<th>HPC</th>
<th>Enterprise Virtualization</th>
<th>Hyperscale</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Software</strong></td>
<td>Accelerated Computing Toolkit</td>
<td>GRID 2.0</td>
<td>Hyperscale Suite</td>
</tr>
<tr>
<td><strong>System Tools &amp; Services</strong></td>
<td>Enterprise Services · Data Center GPU Manager · Mesos · Docker</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Accelerators</strong></td>
<td>Tesla K80</td>
<td>Tesla M60, M6</td>
<td>Tesla M40</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Tesla M4</td>
</tr>
</tbody>
</table>
TESLA FOR HYPERSCALE


**HYPERSCALE SUITE**

- **Deep Learning Toolkit**
- **GPU REST Engine**
- **GPU Accelerated FFmpeg**
- **Image Compute Engine**
- **GPU support in Mesos**

**TESLA M40**

POWERFUL
Fastest Deep Learning Performance

**TESLA M4**

LOW POWER
Highest Hyperscale Throughput
NODE DESIGN FLEXIBILITY

NVLINK
HIGH-SPEED GPU INTERCONNECT
UNIFIED MEMORY: SIMPLER & FASTER WITH NVLINK

Traditional Developer View

- System Memory (Blue)
- GPU Memory (Green)

Developer View With Unified Memory

- Unified Memory

Developer View With Pascal & NVLink

- Unified Memory
  - Share Data Structures at CPU Memory Speeds, not PCIe speeds
  - Oversubscribe GPU Memory

NVLink
MOVE DATA WHERE IT IS NEEDED FAST

Accelerated Communication

- GPU Direct P2P
- Multi-GPU Scaling
- Fast GPU Communication
- Fast GPU Memory Access
- GPU Direct RDMA
- Fast Access to other Nodes
- Eliminate CPU Latency
- Eliminate GPU Bottleneck

- NVLINK
- 2x App Performance
- 5x Faster Than PCIe
- Fast Access to System Memory
ACCELERATORS SURGE IN WORLD’S TOP SUPERCOMPUTERS

100+ accelerated systems now on Top500 list

1/3 of total FLOPS powered by accelerators

NVIDIA Tesla GPUs sweep 23 of 24 new accelerated supercomputers

Tesla supercomputers growing at 50% CAGR over past five years
NEXT-GEN SUPERCOMPUTERS ARE GPU-ACCELERATED

U.S. Dept. of Energy
Pre-Exascale Supercomputers for Science

NOAA
New Supercomputer for Next-Gen Weather Forecasting

IBM Watson
Breakthrough Natural Language Processing for Cognitive Computing
U.S. TO BUILD TWO FLAGSHIP SUPERCOMPUTERS

Powered by the Tesla Platform

100-300 PFLOPS Peak
10x in Scientific App Performance
IBM POWER9 CPU + NVIDIA Volta GPU
NVLink High Speed Interconnect
40 TFLOPS per Node, >3,400 Nodes
2017

Major Step Forward on the Path to Exascale
TESLA PLATFORM LEADS IN EVERY WAY

PROCESSOR

INTERCONNECT

SOFTWARE

ECOSYSTEM

OpenACC
NVIDIA CUDA
ParaView
torch
IBM
TensorFlow

Scaleable Molecular Dynamics
TESLA PLATFORM FOR HPC
"Accelerators Will Be Installed in More than Half of New Systems"

Source: Top 6 predictions for HPC in 2015

"In 2014, NVIDIA enjoyed a dominant market share with 85% of the accelerator market."
370 GPU-Accelerated Applications

www.nvidia.com/appscatalog
70% OF TOP HPC APPS ACCELERATED

INTERSECT360 SURVEY OF TOP APPS

Top 10 HPC Apps
90% Accelerated

Top 50 HPC Apps
70% Accelerated

Top 25 Apps in Survey

GROMACS
SIMULIA Abaqus
NAMD
AMBER
ANSYS Mechanical
Exelis IDL
MSC NASTRAN
ANSYS Fluent
WRF
VASP
OpenFOAM
CHARMM
Quantum Espresso
LAMMPS
NWChem
LS-DYNA
Schroedinger
Gaussian
GAMESS
ANSYS CFX
Star-CD
CCSM
COMSOL
Star-CCM+
BLAST

= All popular functions accelerated
= Some popular functions accelerated
= In development
= Not supported
# TESLA PRODUCTS FOR TARGET SEGMENTS

<table>
<thead>
<tr>
<th>Market Segment</th>
<th>Key Feature</th>
<th>Recommendation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supercomputing, Higher Ed. Research</td>
<td>• Double Precision Perf.</td>
<td>K80, K40</td>
</tr>
<tr>
<td>Oil &amp; Gas</td>
<td>• Memory Bandwidth</td>
<td>K80</td>
</tr>
<tr>
<td>Deep Learning Training</td>
<td>• 12GB Memory per GPU</td>
<td>K80, M40</td>
</tr>
<tr>
<td></td>
<td>• Single Precision Perf. per GPU</td>
<td></td>
</tr>
<tr>
<td>Accelerated Virtual Desktop</td>
<td>• # of Concurrent Connected Users</td>
<td>M60, M6</td>
</tr>
<tr>
<td>Hyperscale</td>
<td>• Low power, Single Precision</td>
<td>M4</td>
</tr>
<tr>
<td></td>
<td>• Small form factor</td>
<td></td>
</tr>
</tbody>
</table>
TESLA K80
World’s Fastest Accelerator for HPC & Data Analytics

5x Faster
AMBER Performance

Simulation Time from 1 Month to 1 Week

Dual CPU Server

Tesla K80 Server

# of Days

CUDA Cores
2496

Peak DP
1.9 TFLOPS

Peak DP w/ Boost
2.9 TFLOPS

GDDR5 Memory
24 GB

Bandwidth
480 GB/s

Power
300 W

GPU Boost
Dynamic

CPU: E5-2698v3 @ 2.30GHz. 64GB System Memory, CentOS 6.2
TESLA K80: 10X FASTER ON REAL-WORLD APPS

CPU: 12 cores, E5-2697v2 @ 2.70GHz, 64GB System Memory, CentOS 6.2
GPU: Single Tesla K80, Boost enabled
TESLA K80 BOOSTS DATA CENTER THROUGHPUT

CPU: Dual E5-2698 v3@2.3GHz, 64GB System Memory, CentOS 6.2
GPU: Single Tesla K80, Boost enabled
CPU versus GPU architecture
LOW LATENCY OR HIGH THROUGHPUT?

CPU

Optimized for low-latency access to cached data sets

Control logic for out-of-order and speculative execution

10’s of threads

GPU

Optimized for data-parallel, throughput computation

Architecture tolerant of memory latency

Massive fine grain threaded parallelism

More transistors dedicated to computation

10000’s of threads
**GPU ARCHITECTURE: TWO MAIN COMPONENTS**

- **Global memory**
  - Analogous to RAM in a CPU server
  - Accessible by both GPU and CPU
  - Currently up to 12 GB per GPU
  - Bandwidth currently up to ~288 GB/s (Tesla products)
  - ECC on/off (Quadro and Tesla products)

- **Streaming Multiprocessors (SMs)**
  - Perform the actual computations
  - Each SM has its own:
    - Control units, registers, execution pipelines, caches
GPU ARCHITECTURE

- SM-0
- SM-1
- SM-N
- GPU L2
- GPU DRAM
- SYSTEM MEMORY
GPU MEMORY HIERARCHY

- SM-0: Registers, L1, SMEM
- SM-1: Registers, L1, SMEM
- SM-N: Registers, L1, SMEM

~ 1 TB/S

~ 150 GB/S

Global Memory
NVIDIA Technology Solves Memory Bandwidth Challenges

- Shared Memory: 1.3 TB/s
- L2 Cache: 280 GB/s
- GPU Memory: 177 GB/s
- Register: 10.8 TB/s

PCI-Express 6.4 GB/s

NVIDIA GPU

SCIENTIFIC COMPUTING CHALLENGE: MEMORY BANDWIDTH
Architecture

- 7.1B Transistors
- 15 SMX units
- > 1 TFLOP FP64
- 1.5 MB L2 Cache
- 384-bit GDDR5
Functional Units = CUDA cores
- 192 SP FP operations/clock
- 64 DP FP operations/clock
Register file (256KB)
Shared memory (16-48KB)
L1 cache (16-48KB)
Read-only cache (48KB)
Constant cache (8KB)
SIMT EXECUTION MODEL

Thread: sequential execution unit

- All threads execute same sequential program
- Threads execute in parallel

Thread Block: a group of threads

- Threads within a block can cooperate
  - Light-weight synchronization
  - Data exchange

Grid: a collection of thread blocks

- Thread blocks do not synchronize with each other
- Communication between blocks is expensive
SIMT EXECUTION MODEL

Threads are executed by CUDA Cores

Thread blocks are executed on multiprocessors

Thread blocks do not migrate

Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)

A kernel is launched as a grid of thread blocks
SIMT EXECUTION MODEL

Threads are organized into groups of 32 threads called “warps”

All threads within a warp execute the same instruction simultaneously
1. Copy input data from CPU memory/NIC to GPU memory
1. Copy input data from CPU memory/NIC to GPU memory
2. Load GPU program
1. Copy input data from CPU memory/NIC to GPU memory
2. Load GPU program and execute
3. Copy results from GPU memory to CPU memory/NIC
We must expose enough parallelism to saturate the device.

Accelerator threads are slower than CPU threads.

Accelerators have orders of magnitude more threads.

**Fine-grained** parallelism is good.

**Coarse-grained** parallelism is bad.
Express as much parallelism as possible: SMXs (Kepler) are wider than SMs (Fermi)

Tetris (tile = warp_instr):
- Issues 4 warp_instrs.
- Executes up to 10 warps = 320 threads.
- Warp_instrs. are symmetric and executed all in one cycle.

Example: Kernel with blocks of 384 threads (12 warps).

<table>
<thead>
<tr>
<th>Block 0:</th>
<th>Block 1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub</td>
<td></td>
</tr>
<tr>
<td>fmadd</td>
<td></td>
</tr>
<tr>
<td>fdiv</td>
<td></td>
</tr>
<tr>
<td>load</td>
<td></td>
</tr>
<tr>
<td>sqrt</td>
<td></td>
</tr>
</tbody>
</table>

G80: Takes 4 cycles for executing each warp_instrs.

**Fermi:**
- Issues 2.
- Executes up to 5.

**Kepler:**
- Issues 4 warps x 2 instructions.
- Executes up to 16 warp_instrs.
  (up to 512 functional units in parallel)

Color code:
- Yellow: for instructions using “int”.
- Orange: for instrs. using “float”.
- Red: “double”.
- Green: “load/store”.
- Blue: “log/sqrt…”
- The player is the GPU scheduler!
  You can rotate moving pieces if there are no data dependencies.

Executes up to 10 warp_instrs.

**G80:** 16 U.F.

**SM in Fermi:** 100 functional units

**SM in Kepler:** 512 functional units

32 SFU
32 LD/ST
64 DP FPU

6x32 = 192 ALUs
192 SP FPU
Thread Level Parallelism (TLP) and Instruction Level Parallelism (ILP)

- Increase parallelism horizontally via TLP: More **concurrent warps** (larger blocks and/or more active blocks per SMX).

- Increase parallelism vertically via ILP: Using more independent instructions.

- **SMXs** can leverage available ILP interchangeably with TLP.
- Sometimes is easier to increase ILP than TLP (for example, a small loop unrolling)

- **We need ILP** for attaining a high IPC (Instrs. Per Cycle).
Kepler GPUs can hold together all forms of parallelism. Example: K40.

Imagine a 3D tetris with 15 boxes and up to 64 pieces falling down simultaneously on each of them.
BEST PRACTICES

Optimize Data Locality: GPU

Minimize data transfers between CPU and GPU
BEST PRACTICES

Optimize Data Locality: SM

Minimize redundant accesses to L2 and DRAM

Store intermediate results in registers instead of global memory

Use shared memory for data frequently used within a thread block

Use `const __restrict__` to take advantage of read-only cache
TESLA PLATFORM FOR DEVELOPERS
TESLA FOR SIMULATION

LIBRARIES

DIRECTIVES

LANGUAGES

ACCELERATED COMPUTING TOOLKIT

TESLA ACCELERATED COMPUTING
DROP-IN ACCELERATION WITH GPU LIBRARIES

5x-10x speedups out of the box

Automatically scale with multi-GPU libraries (cuBLAS-XT, cuFFT-XT, AmgX,...)

75% of developers use GPU libraries to accelerate their application
“DROP-IN” ACCELERATION: NVBLAS

Automatic Speedup for “R” application

```r
LD_PRELOAD=/usr/local/cuda/lib64/libnvblas.so R
A <- matrix(rnorm(4096*4096), nrow=4096, ncol=4096)
B <- matrix(rnorm(4096*4096), nrow=4096, ncol=4096)
system.time(C <- A %*% B)
```

User system elapsed time: 0.348 0.142 0.289

NO CODE CHANGE REQUIRED

Use in any app that uses standard BLAS3: R, Octave, Scilab, etc.

GPUWhiz
*Automatically detect and replace calls to GEMM with cuBLAS
No application code changes
Good for existing libraries where BLAS heavily used.
OpenACC
Simple | Powerful | Portable
Fueling the Next Wave of Scientific Discoveries in HPC

main()
{
    <serial code>
    #pragma acc kernels
    //automatically runs on GPU
    {
        <parallel code>
    }
}

University of Illinois
PowerGrid- MRI Reconstruction

70x Speed-Up
2 Days of Effort

RIKEN Japan
NICAM- Climate Modeling

7-8x Speed-Up
5% of Code Modified

8000+
Developers
using OpenACC

http://www.cray.com/sites/default/files/resources/OpenACC_213462.12_OpenACC_Cosmo_CS_FNL.pdf
http://www.openacc.org/content/experiences-porting-molecular-dynamics-code-gpus-cray-xk7
LS-DALTON
Large-scale Application for Calculating High-accuracy Molecular Energies

OpenACC makes GPU computing approachable for domain scientists. Initial OpenACC implementation required only minor effort, and more importantly, no modifications of our existing CPU implementation.

Janus Juul Eriksen, PhD Fellow
qLEAP Center for Theoretical Chemistry, Aarhus University

Minimal Effort

<table>
<thead>
<tr>
<th>Lines of Code Modified</th>
<th># of Weeks Required</th>
<th># of Codes to Maintain</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;100 Lines</td>
<td>1 Week</td>
<td>1 Source</td>
</tr>
</tbody>
</table>

Big Performance

LS-DALTON CCSD(T) Module

Benchmarked on Titan Supercomputer (AMD CPU vs Tesla K20X)

<table>
<thead>
<tr>
<th></th>
<th>Speedup vs CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alanine-1</td>
<td>12.0x</td>
</tr>
<tr>
<td>Alanine-2</td>
<td>8.0x</td>
</tr>
<tr>
<td>Alanine-3</td>
<td>4.0x</td>
</tr>
</tbody>
</table>

Alanine-1
13 Atoms

Alanine-2
23 Atoms

Alanine-3
33 Atoms
OPENACC DELIVERS TRUE PERFORMANCE PORTABILITY

Paving the Path Forward: Single Code for All HPC Processors

Application Performance Benchmark

- CPU: MPI + OpenMP
- CPU: MPI + OpenACC
- CPU + GPU: MPI + OpenACC

### Speedup vs Single CPU Core

<table>
<thead>
<tr>
<th>Application</th>
<th>CPU: MPI + OpenMP</th>
<th>CPU: MPI + OpenACC</th>
<th>CPU + GPU: MPI + OpenACC</th>
</tr>
</thead>
<tbody>
<tr>
<td>359.miniGhost (MANTEVO)</td>
<td>4.1x</td>
<td>4.3x</td>
<td>7.6x</td>
</tr>
<tr>
<td>NEMO (CLIMATE &amp; OCEAN)</td>
<td>5.2x</td>
<td>5.3x</td>
<td>11.9x</td>
</tr>
<tr>
<td>CLOVERLEAF (PHYSICS)</td>
<td>7.1x</td>
<td>7.1x</td>
<td>30.3x</td>
</tr>
</tbody>
</table>

359.miniGhost: CPU: Intel Xeon E5-2698 v3, 2 sockets, 32 cores total, GPU: Tesla K80- single GPU

NEMO: Each socket CPU: Intel Xeon E5-2698 v3, 16 cores; GPU: NVIDIA K80 both GPUs

CLOVERLEAF: CPU: Dual socket Intel Xeon CPU E5-2690 v2, 20 cores total, GPU: Tesla K80 both GPUs
SPEC ACCEL ON HASWELL AND TESLA K80

**System Information:**
Supermicro SYS-2028GR-TRT
CPU: Intel Xeon E5-2698 v3, 2 sockets, 32 cores, HT disabled
GPU: NVIDIA Tesla K80 (single GPU)
OS: CentOS 6.6
Compiler: PGI 15.9

PGI 15.9 OpenACC Multicore and K80 results from SPEC ACCEL™ measured Sept 2015. These are SPEC® Estimates. SPEC® and the benchmark name SPEC ACCEL™ are registered trademarks of the Standard Performance Evaluation Corporation.
INTRODUCING THE NEW OPENACC TOOLKIT

Free Toolkit Offers Simple & Powerful Path to Accelerated Computing

PGI Compiler
Free OpenACC compiler for academia

NVProf Profiler
Easily find where to add compiler directives

GPU Wizard
Identify which GPU libraries can jumpstart code

Code Samples
Learn from examples of real-world algorithms

Documentation
Quick start guide, Best practices, Forums

http://developer.nvidia.com/openacc
<table>
<thead>
<tr>
<th>DATE</th>
<th>COURSE</th>
<th>REGION</th>
</tr>
</thead>
<tbody>
<tr>
<td>March 2016</td>
<td>Intro to Performance Portability with OpenACC</td>
<td>China</td>
</tr>
<tr>
<td>March 2016</td>
<td>Intro to Performance Portability with OpenACC</td>
<td>India</td>
</tr>
<tr>
<td>May 2016</td>
<td>Advanced OpenACC</td>
<td>Worldwide</td>
</tr>
<tr>
<td>September 2016</td>
<td>Intro to Performance Portability with OpenACC</td>
<td>Worldwide</td>
</tr>
</tbody>
</table>

Registration page: https://developer.nvidia.com/openacc-courses

Self-paced labs: http://nvidia.qwiklab.com
SAXPY - SINGLE PRECISION A*X PLUS Y

**SAXPY in C**

```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);
```

**SAXPY in Fortran**

```fortran
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

...!

! Perform SAXPY on N elements
call saxpy(N, 2.0, x, y)
...```
**SAXPY - SINGLE PRECISION A*X PLUS Y IN OPENMP - CPU**

### SAXPY in C

```c
void saxpy(int n, float a,
    float *x, float *y)
{
    #pragma omp parallel for
    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);
}
```

### SAXPY in Fortran

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

...!
Perform SAXPY on N elements
```

```fortran
call saxpy(N, 2.0, x, y)
...!
```
SAXPY - SINGLE PRECISION A*X PLUS Y IN OPENACC - CPU & ACCELERATOR

**SAXPY in C**

```c
void saxpy(int n, float a, 
            float *x, float *y) 
{
#pragma acc parallel loop
  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);
```

**SAXPY in Fortran**

```fortran
subroutine saxpy(n, a, x, y)
  real :: x(*), y(*), a
  integer :: n, i

  !$acc parallel loop!
  do i=1,n
    y(i) = a*x(i)+y(i)
  enddo

  !$acc end parallel
end subroutine saxpy

int N = 1<<20;

! Perform SAXPY on N elements
saxpy(N, 2.0, x, y)
```

...
SAXPY - SINGLE PRECISION A*X PLUS Y IN OPENMP - ACCELERATOR (GPU)

**SAXPY in C**

```c
void saxpy(int n, float a, 
    float *x, float *y)
{
    #pragma omp target teams \ 
    distribute parallel for
    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);
```

**SAXPY in Fortran**

```fortran
subroutine saxpy(n, a, x, y)
    real :: x(*), y(*), a
    integer :: n, i
    !$omp target teams &
    !$omp distribute parallel do
    do i=1,n
        y(i) = a*x(i)+y(i)
    enddo
    !$omp end target teams &
    !$omp distribute parallel do
end subroutine saxpy
...
! Perform SAXPY on N elements
call saxpy(N, 2.0, x, y)
...```
Portable parallelism is easier to maintain.

```c
if defined(CPU)
#pragma omp parallel for simd
elif defined(MIC)
#pragma omp target teams distribute \ parallel for simd
elif defined(OMP_GPU)
#pragma omp target teams distribute \ parallel for schedule(static,1)
elif defined(SOMETHING_ELSE)
#pragma omp target ...
endif
for(int i = 0; i < N; i++)
```

```c
#pragma acc parallel loop
For(int I = 0; I < N; i++)
```
SINGLE EXAMPLE ABOUT HOW TO EXPRESS PARALLELISM AND DATA LOCALITY USING COMPILED DIRECTIVES LANGUAGES USING A GPU ACCELERATOR

1. Identify Parallelism
2. Express Parallelism
3. Express Data Locality
4. Optimize

Data must be transferred between CPU and GPU memories.
EXAMPLE: JACOBI ITERATION

- Iteratively converges to correct value (e.g. Temperature), by computing new values at each point from the average of neighboring points.
  - Common, useful algorithm
  - Example: Solve Laplace equation in 2D: \( \nabla^2 f(x, y) = 0 \)

\[
A_{k+1}(i, j) = \frac{A_k(i-1, j) + A_k(i+1, j) + A_k(i, j-1) + A_k(i, j+1)}{4}
\]
JACOBI ITERATION: C CODE

```c
while (err > tol && iter < iter_max) {
    err=0.0;

    for( int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }

    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
}
iter++;
```
while ( err > tol && iter < iter_max ) {
    err=0.0;
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++) {
            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
JACOBI ITERATION: OPENMP C CODE FOR CPU

```c
while (err > tol && iter < iter_max) {
    err=0.0;

#pragma omp parallel for shared(m, n, Anew, A) reduction(max:err)
    int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                                 A[j-1][i] + A[j+1][i]);
            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }

#pragma omp parallel for shared(m, n, Anew, A)
    int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
            A[j][i] = Anew[j][i];
        }
    }

   [60][60] = iter;
}
```

Parallelize loop across CPU threads

Identify Parallelism → Express Parallelism → Express Data Locality → Optimize
JACOBI ITERATION: OPENACC C CODE - CPU&GPU

err > tol && iter < iter_max ) {
    err=0.0;

#pragma acc parallel loop reduction(max:err)
    int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++) {

            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] + 
                              A[j-1][i] + A[j+1][i]);

            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }

#pragma acc parallel loop
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}
$ pgcc -acc -ta=nvidia:5.5,kepler -Minfo=accel -o laplace2d_acc laplace2d.c

main:

56, Accelerator kernel generated

57, #pragma acc loop gang /* blockIdx.x */
59, #pragma acc loop vector(256) /* threadIdx.x */

56, Generating present_or_copyout(Anew[1:4094][1:4094])
Generating present_or_copyin(A[0:][0:]))
Generating NVIDIA code
Generating compute capability 3.0 binary

59, Loop is parallelizable

63, Max reduction generated for error

68, Accelerator kernel generated

69, #pragma acc loop gang /* blockIdx.x */
71, #pragma acc loop vector(256) /* threadIdx.x */

68, Generating present_or_copyin(Anew[1:4094][1:4094])
Generating present_or_copyout(A[1:4094][1:4094])
Generating NVIDIA code
Generating compute capability 3.0 binary

71, Loop is parallelizable
Why is OpenACC so much slower?
$ nvprof ./laplace2d_acc

Jacobi relaxation Calculation: 4096 x 4096 mesh

--10619-- NVPROF is profiling process 10619, command: ./laplace2d_acc

0, 0.250000
100, 0.002397
200, 0.001204
300, 0.000804
400, 0.000603
500, 0.000483
600, 0.000403
700, 0.000345
800, 0.000302
900, 0.000269

total: 134.259326 s

--10619-- Profiling application: ./laplace2d_acc

--10619-- Profiling result:

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>49.59%</td>
<td>44.0095s</td>
<td>17000</td>
<td>2.5888ms</td>
<td>864ns</td>
<td>2.9822ms</td>
<td>[CUDA memcpy HtoD]</td>
</tr>
<tr>
<td>45.06%</td>
<td>39.9921s</td>
<td>17000</td>
<td>2.3525ms</td>
<td>2.4960us</td>
<td>2.7687ms</td>
<td>[CUDA memcpyDtoH]</td>
</tr>
<tr>
<td>2.95%</td>
<td>2.61622s</td>
<td>1000</td>
<td>2.6162ms</td>
<td>2.6044ms</td>
<td>2.6319ms</td>
<td>main_56_gpu</td>
</tr>
<tr>
<td>2.39%</td>
<td>2.11884s</td>
<td>1000</td>
<td>2.1188ms</td>
<td>2.1023ms</td>
<td>2.1374ms</td>
<td>main_68_gpu</td>
</tr>
<tr>
<td>0.01%</td>
<td>12.431ms</td>
<td>1000</td>
<td>12.430us</td>
<td>12.192us</td>
<td>12.736us</td>
<td>main_63_gpu_red</td>
</tr>
</tbody>
</table>
Excessive Data Transfers

while ( err > tol && iter < iter_max )
{
    err=0.0;
    #pragma acc parallel loop reduction(max:err)
    for( int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }
}

A, Anew resident on host
These copies happen every iteration of the outer while loop!

A, Anew resident on host
Copy

A, Anew resident on accelerator

Copy

A, Anew resident on accelerator

=> Need to use directive to control data location and transfers
Jacobi Iteration: OpenACC C Code

```c
#pragma acc data copy(A) create(Anew)
while ( err > tol && iter < iter_max ) {
  err=0.0;

#pragma acc parallel loop reduction(max:err)
  for( int j = 1; j < n-1; j++ ) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                            A[j-1][i] + A[j+1][i]);

      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }

#pragma acc parallel loop
  for( int j = 1; j < n-1; j++ ) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

Copy A to/from the accelerator only when needed.
Create Anew as a device temporary.
PGI 15.1: OpenACC directive-based data movement vs OpenACC w/CUDA 6.5 Unified Memory on Kepler

Features:
- Fortran ALLOCATE and C/C++ malloc/calloc/new can automatically use CUDA Unified Memory
- No explicit transfers needed for dynamic data (or allowed, for now)

Limitations:
- Supported only for dynamic data
- Program dynamic memory size is limited by UM data size
- UM data motion is synchronous
- Can be unsafe
INDEPENDENT CLAUSE

```
while ( err > tol && iter < iter_max ) {
    err=0.0;

    #pragma acc kernels
    {
        #pragma acc loop independent
        for( int j = 1; j < n-1; j++ ) {
            for(int i = 1; i < m-1; i++) {

                Anew[j*m+i] = 0.25 * ( A[j*m+i+1] + A[j*m+i-1] + 
                                         A[(j-1)*m+i] + A[(j+1)*m+i] );

                err = max(err, abs(Anew[j*m+i] - A[j*m+i]));
            }
        }
    }

    #pragma acc loop independent
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j*m+i] = Anew[j*m+i];
        }
    }

    iter++;
}
```
OPENACC AND CUDA UNIFIED MEMORY

BUILDING THE CODE

$ pgcc -fast -acc -ta=tesla:managed -Minfo=all laplace2d.c
main:
  83, Generating copyout(Anew[:])
  Generating copy(A[:])
  86, Loop is parallelizable
  87, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
    86, #pragma acc loop gang /* blockIdx.y */
    87, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    92, Max reduction generated for error
  97, Loop is parallelizable
  98, Loop is parallelizable
    Accelerator kernel generated
    Generating Tesla code
    97, #pragma acc loop gang /* blockIdx.y */
    98, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
OPENACC AND CUDA UNIFIED MEMORY

PERFORMANCE RESULTS

Speed-up (Higher is Better)

Intel Xeon E5-2698 v3 @ 2.30GHz (Haswell) vs. NVIDIA Tesla K40 (Kepler)

Socket/Socket: 6.86X

1.0 2.0 3.5 4.3 4.4 30.2

Single Thread 2 Threads 4 Threads 6 Threads 8 Threads OpenACC
PROGRAMMING LANGUAGES

**Numerical analytics**
- MATLAB, Mathematica, LabVIEW, Scilab, Octave

**Fortran**
- OpenACC, CUDA Fortran

**C**
- OpenACC, CUDA C

**C++**
- Thrust, CUDA C++, KOKKOS, RAJA, HEMI, OCCA

**Python**
- PyCUDA, Copperhead, Numba, Numbapro

**JAVA,C#**
- GPU.NET, Hybridizer (Altimesh), JCUDA, CUDA4J
COMPILE PYTHON FOR PARALLEL ARCHITECTURES

- Anaconda Accelerate from Continuum Analytics
  - NumbaPro array-oriented compiler for Python & NumPy
  - Compile for CPUs or GPUs (uses LLVM + NVIDIA Compiler SDK)

- Fast Development + Fast Execution: Ideal Combination

Free Academic License
http://continuum.io
CUDA

Super Simplified Memory Management Code

**CPU Code**

```c
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);
  fread(data, 1, N, fp);
  qsort(data, N, 1, compare);
  use_data(data);
  free(data);
}
```

**CUDA 6 Code with Unified Memory**

```c
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);
  fread(data, 1, N, fp);
  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();
  use_data(data);
  cudaFree(data);
}
```
Forward Wave Propagation Throughput

- ~1.8x speedup, every 18 – 24 months
- Model size: \((512)^3\)
- Isotropic: 8\(^{th}\) order space, 2\(^{nd}\) order time
- Two versions of code
  - Tuned
    - Same source, Fermi & Kepler generations
    - Not applicable to M1060
  - Simple
    - Blue-Green over 3 gen.

- 2008 (Tesla M1060): 3,300 Mcells/s
- 2010 (Tesla M2090): 6,100 Mcells/s
- 2012 (Tesla K10): 11,000 Mcells/s
- 2014 (Tesla K80): 18,700 Mcells/s
MORE C++ PARALLEL FOR LOOPS

GPU Lambdas Enable Custom Parallel Programming Models

Kokkos

Kokkos::parallel_for(N, KOKKOS_LAMBDA (int i) {
    y[i] = a * x[i] + y[i];
});

https://github.com/kokkos

RAJA

RAJA::forall<cuda_exec>(0, N, [=] __device__ (int i) {
    y[i] = a * x[i] + y[i];
});

https://e-reports-ext.llnl.gov/pdf/782261.pdf

Hemi

Hemi::parallel_for(0, N, [=] HEMI_LAMBDA (int i) {
    y[i] = a * x[i] + y[i];
});

http://github.com/harrism/hemi
THRUST LIBRARY
Programming with algorithms and policies today

Bundled with NVIDIA’s CUDA Toolkit

Supports execution on GPUs and CPUs

Ongoing performance & feature improvements

Functionality beyond Parallel

Thrust Sort Speedup
CUDA 7.0 vs. 6.5 (32M samples)

From CUDA 7.0 Performance Report.
Run on K40m, ECC ON, input and output data on device
Performance may vary based on OS and software versions, and motherboard configuration
PortaAle, High-level Parallel Code TODAY

- Thrust library allows the same C++ code to target both:
  - NVIDIA GPUs
  - x86, ARM and POWER CPUs

- Thrust was the inspiration for a proposal to the ISO C++ Committee

Technical Specification for C++ Extensions for Parallelism
Published as ISO/IEC TS 19570:2015, July 2015

Draft available online


We’ve proposed adding this to C++17

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0024r0.html
CUDA Architecture
- Expose GPU parallelism for general-purpose computing
- Retain performance

CUDA C/C++
- Based on industry-standard C/C++
- Small set of extensions to enable heterogeneous programming
- Straightforward APIs to manage devices, memory etc.

This session introduces CUDA C/C++
Introduction to CUDA C/C++

• What will you learn in this session?
  – Start from “Hello World!”
  – Write and launch CUDA C/C++ kernels
  – Manage GPU memory
  – Manage communication and synchronization
Prerequisites

• You (probably) need experience with C or C++

• You don’t need GPU experience

• You don’t need parallel programming experience

• You don’t need graphics experience
HELLO WORLD!

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices
Heterogeneous Computing

- **Host**: The CPU and its memory (host memory)
- **Device**: The GPU and its memory (device memory)
Some Terminology

- **Kernel** – A function which runs on the GPU
  - A kernel is launched on a grid of thread blocks.
  - The grid and block size are called the launch configuration.

- **Global Memory** – GPU’s on-board DRAM

- **Shared Memory** – On-chip fast memory local to a thread block
CUDA Execution Model

- **Thread**: Sequential execution unit
  - All threads execute same sequential program
  - Threads execute in parallel

- **Thread Block**: a group of threads
  - Executes on a single Streaming Multiprocessor (SM)
  - Threads within a block can cooperate
    - Light-weight synchronization
    - Data exchange

- **Grid**: a collection of thread blocks
  - Thread blocks of a grid execute across multiple SMs
  - Thread blocks do not synchronize with each other
  - Communication between blocks is expensive
IDs and Dimensions

- A kernel is launched as a grid of blocks of threads

- Built-in variables:
  - threadIdx
  - blockIdx
  - blockDim
  - gridDim
Heterogeneous Computing

```cpp
#include <iostream>
#include <algorithm>
using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16
__global__
void stencil_1d(
  int* in,
  int* out)
{
  __shared__
  int temp[BLOCK_SIZE + 2 * RADIUS];

  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
  }

  __syncthreads();

  // Apply the stencil
  int result = 0;
  for (int offset = -RADIUS; offset <= RADIUS; offset++)
    result += temp[lindex + offset];

  // Store the result
  out[gindex] = result;
}

void fill_ints(int* x, int n) {
  fill_n(x, n, 1);
}

int main(void) {
  int* in, *out;
  // host copies of a, b, c
  int* d_in, *d_out;
  // device copies of a, b, c
  int size = (N + 2*RADIUS) * sizeof(int);
  // Alloc space for host copies and setup values
  in  = (int*) malloc(size);
  fill_ints(in, N + 2*RADIUS);
  out = (int*) malloc(size);
  fill_ints(out, N + 2*RADIUS);
  // Alloc space for device copies
  cudaMalloc((void**)&d_in, size);
  cudaMalloc((void**)&d_out, size);
  // Copy to device
  cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);
  // Launch kernel on GPU
  stencil_1d<<<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);
  // Copy result back to host
  cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);
  // Cleanup
  free(in); free(out);
  cudaFree(d_in);
  cudaFree(d_out);
  return 0;
}
```
Memory Management

- Host and device memory are separate entities
  - *Device pointers* point to GPU memory
    May be passed to/from host code
    May *not* be dereferenced in host code
  - *Host pointers* point to CPU memory
    May be passed to/from device code
    May *not* be dereferenced in device code

- Simple CUDA API for handling device
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`
  - `cudaMallocManaged()` for Unified Memory
Multiple Memory Spaces

- Memory is allocated with locality
  - `cudaMalloc(&d_ptr)` – device
    - `d_ptr` cannot be dereferenced from the CPU (before CUDA 6.0)
  - `Ptr = Malloc(); ptr=new, cudaMallocHost(&ptr)` - CPU
    - `Ptr` cannot be dereferenced from the GPU (before CUDA 6.0)

- Zero Copy and GPUDirect allow you to circumvent (functionally)
  - Future architectures will improve.

- As of CUDA 4.0, virtual address range of CPU and GPU are unique
  - Can determine where the target of a pointer lives
- From CUDA 6.0 Unified Memory could be used
Unified Memory (CUDA 6.0+)
Dramatically Lower Developer Effort

Developer View Today

System Memory

GPU Memory

Developer View With Unified Memory

Unified Memory
void sortfile(FILE *fp, int N) {
    char *data;
    data = (char *)malloc(N);
    fread(data, 1, N, fp);
    qsort(data, N, 1, compare);
    use_data(data);
    free(data);
}

void sortfile(FILE *fp, int N) {
    char *data;
    cudaMallocManaged(&data, N);
    fread(data, 1, N, fp);
    qsort<<<...>>>(data, N, 1, compare);
    cudaDeviceSynchronize();
    use_data(data);
    cudaFree(data);
}
Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory
int main(void) {
    printf("Hello World!\n");
    return 0;
}

- Standard C that runs on the host

- NVIDIA compiler (nvcc) can be used to compile programs with no device code

Output:

```
$ nvcc hello_world.cu
$ a.out
Hello World!
$
```
Hello World! with Device Code

```c
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

- Two new syntactic elements...
Hello World! with Device Code

```c
__global__ void mykernel(void) {
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code

- `nvcc` separates source code into host and device components
  - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler
    - `gcc`, `cl.exe`
Hello World! with Device Code

mykernel<<<1,1>>>();

• Triple angle brackets mark a call from host code to device code
  – Also called a “kernel launch”
  – We’ll return to the parameters (1,1) in a moment

• That’s all that is required to execute a function on the GPU!
Hello World! with Device Code

```c
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

- mykernel() does nothing, somewhat anticlimactic!

Output:

```
$ nvcc hello.cu
$ a.out
Hello World!
$ 
```
Parallel Programming in CUDA C/C++

- But wait... GPU computing is about massive parallelism!
- We need a more interesting example...
- We’ll start by adding two integers and build up to vector

```
\[ \begin{align*}
   a + b &= c \\
\end{align*} \]
```
Vector Add: GPU’s Hello World

• GPU is parallel computation oriented.
  – Vector add is a very simple parallel algorithm.

• Problem: C = A + B
  – C, A, B are length N vectors

```c
void vecAdd(int n, float * a,
            float * b, float * c)
{
    for(int i=0; i<n; i++)
    {
        c[i] = a[i] + b[i];
    }
}
```
Vector Add: GPU’s Hello World

• GPU is *parallel computation* oriented.
  – Vector add is a very simple parallel algorithm.

• Problem: $C = A + B$
  – $C$, $A$, $B$ are length $N$ vectors

```c
void main()
{
  int N = 1024;
  float * a, *b, *c;
  a = (float*)malloc(N*sizeof(float));
  b = (float*)malloc(N*sizeof(float));
  c = (float*)malloc(N*sizeof(float));
  memset(c, 0, N*sizeof(float));
  init_rand_f(a, N);
  init_rand_f(b, N);
  vecAdd(N, a, b, c);
}
```

```c
void vecAdd(int n, float * a, float * b, float * c)
{
  for(int i=0; i<n; i++)
  {
    c[i] = a[i] + b[i];
  }
}
```
Moving Computation to the GPU

• **Step 1:** Identify parallelism.
  – Design problem decomposition
• **Step 2:** Write your GPU Kernel
• **Step 3:** Setup the Problem
• **Step 4:** Launch the Kernel
• **Step 5:** Copy results back from GPU

Remember: big font means important
Vector Add

• **Step 1: Parallelize**
  – Identify parallelism.
    • c[i] depends only on a[i] and b[i].
    • c[i] is not used by any other calculation
    • c[i] can be computed in parallel

  – **Assign Units of Work**
    • Each thread will compute one element of c.
    • Will use a 1D grid of 1D threadblocks.

```c
void vecAdd(int n, float * a, float * b, float * c)
{
    for(int i=0; i<n; i++)
    {
        c[i] = a[i] + b[i];
    }
}
```
Historical Note

• Origins of GPU Computing:
  – Vertex Shader: Single program, all vertices in a 3D scene
  – Fragment Shader: single program, all pixels in a ‘fragment’

• The Natural Approach:
  – Use one thread per output item
  – Good starting point if problem supports it

• CUDA and GPU Computing Have Evolved
  – Much more capable than this model
Parallelization of VecAdd

| Thread | 0  | 1  | 2  | ... | 255 | 256 | ... | 511 | 512 | ... | 767 | ... | N   |
|--------|----|----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
RUNNING IN PARALLEL
BLOCKS & THREADS

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices
Parallelization of VecAdd

Thread Block 0

0    1    2    ...    255


Thread Block 1

256  ...  511

a[256]  a[511]
b[256]  b[511]
c[256]  c[511]

Thread Block 2

512  ...  767

a[512]  a[767]
b[512]  b[767]
c[512]  c[767]

...  N

N

 inputs
 outputs

Thread
IDs and Dimensions

• A kernel is launched as a grid of blocks of threads
  • blockIdx and threadIdx are 3D
  • We showed only one dimension (x)

• Built-in variables:
  – threadIdx
  – blockIdx
  – blockDim
  – gridDim
• Built-in variables:
  - threadIdx.\([x \ y \ z]\)
    • thread index within a thread block
  - blockIdx.\([x \ y \ z]\)
    • block index within the grid.
  - blockDim.\([x \ y \ z]\)
    • Number of threads in each block.
  - gridDim.\([x \ y \ z]\)
    • Number of blocks in the grid.
Parallelization of VecAdd

<table>
<thead>
<tr>
<th>Thread Block 0</th>
<th>Thread Block 1</th>
<th>Thread Block 2</th>
</tr>
</thead>
</table>

work index \( i = \text{threadIdx.x} + \text{blockIdx.x} \times \text{blockDim.x} \)

- **Thread**: Index of the thread within a thread block
- **inputs**: Index of the threadblock within the grid
- **outputs**: Number of threads within each block
Vector Add: GPU’s Hello World

• Step 2: Make it a GPU Kernel

Identify this function as something to be run on the GPU.

Protect against invalid access if too many threads are launched.

```c
__global__ void vecAdd(int n, float * a, float * b, float * c)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i < n) {
        c[i] = a[i] + b[i];
    }
}
```
void main() {
  int N = 1024;
  float *a, *b, *c;
  float *devA, *devB, *devC;
  a = (float*)malloc(N*sizeof(float));
  b = (float*)malloc(N*sizeof(float));
  c = (float*)malloc(N*sizeof(float));
  cudaMalloc(&devA, N*sizeof(float));
  cudaMalloc(&devB, N*sizeof(float));
  cudaMalloc(&devC, N*sizeof(float));
  memset(c, 0, N*sizeof(float));
  init_rand_f(a, N);
  init_rand_f(b, N);
  cudaMemcpy(devA, a, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(devB, b, N*sizeof(float), cudaMemcpyHostToDevice);
}
void main()
{
    ...
    vecAdd<<<(N+127)/128, 128>>>(N, devA, devB, devC);
    ...
}

Step 4: Launch the GPU Kernel

call function by name as usual

Angle Brackets: Specify launch configuration for the kernel.

Normal parameter passing syntax. Note that devA, devB, and devC are device pointers. They point to memory allocated on the GPU.

First argument is the number of thread blocks (rounding up)

Second argument is the shape of (i.e., number of threads in) each thread block
void main()
{
  ...
  cudaMemcpy(c, devC, N*sizeof(float), cudaMemcpyDeviceToHost);
  ...
}
Threads vs Blocks

• Distinction is not clear from previous example

• Threads within a block can:
  – Communicate
  – Synchronize

• New example to illuminate this subject.
COOPERATING THREADS

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices
1D Stencil

• Consider applying a 1D stencil to a 1D array of elements
  – Each output element is the sum of input elements within a radius

• If radius is 3, then each output element is the sum of 7 input elements:
Implementing Within a Block

- Each thread processes one output element
  - `blockDim.x` elements per block

- Input elements are read several times
  - With radius 3, each input element is read seven times
Sharing Data Between Threads

• Terminology: within a block, threads share data via shared memory

• Extremely fast on-chip memory, user-managed

• Declare using __shared__, allocated per block

• Data is not visible to threads in other blocks
Implementing With Shared Memory

• Cache data in shared memory
  – **Read** \((\text{blockDim}.x + 2 \times \text{radius})\) input elements from global memory to shared memory
  – **Compute** \(\text{blockDim}.x\) output elements
  – **Write** \(\text{blockDim}.x\) output elements to global memory
  – Each block needs a **halo** of \(\text{radius}\) elements at each boundary
__global__ void stencil_1d(int *in, int *out) {
__shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
int gindex = threadIdx.x + blockIdx.x * blockDim.x
int lindex = threadIdx.x

// Read input elements into shared memory
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
Stencil Kernel

// Apply the stencil
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
  result += temp[lindex + offset];

// Store the result
out[gindex] = result;
}
Data Race!

- The stencil example will not work...

- Suppose thread 15 reads the halo before thread 0 has fetched it...

```c
temp[lindex] = in[gindex];  // Store at temp[18]
if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS]; // Skipped, threadIdx > RADIUS
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
int result = 0;
result += temp[lindex + 1];  // Load from temp[19]
```
__syncthreads()

- void __syncthreads();

- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards

- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block
__global__ void stencil_1d (int *in, int *out) {
    int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();
Stencil Kernel

// Apply the stencil
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

// Store the result
out[gindex] = result;
Review (1 of 2)

• Launching parallel threads
  – Launch $N$ blocks with $M$ threads per block with
    $\text{kernel}<<<N,M>>>(\ldots)$;
  – Use $\text{blockIdx.x}$ to access block index within grid
  – Use $\text{threadIdx.x}$ to access thread index within block

• Allocate elements to threads:

\[
\text{int } \text{index} = \text{threadIdx.x} + \text{blockIdx.x} * \text{blockDim.x}
\]
Review (2 of 2)

• **Use `__shared__` to declare a variable/array in shared memory**
  – Data is shared between threads in a block
  – Not visible to threads in other blocks

• **Use `__syncthreads()` as a barrier**
  – Use to prevent data hazards
MANAGING THE DEVICE
Coordinating Host & Device

- Kernel launches are **asynchronous**
  - Control returns to the CPU immediately

- CPU needs to synchronize before consuming the results

  `cudaMemcpy()`  
  Blocks the CPU until the copy is complete
  Copy begins when all preceding CUDA calls have completed

  `cudaMemcpyAsync()`  
  Asynchronous, does not block the CPU

  `cudaDeviceSynchronize()`  
  Blocks the CPU until all preceding CUDA calls have completed
Reporting Errors

• All CUDA API calls return an error code (cudaError_t)
  – Error in the API call itself
  OR
  – Error in an earlier asynchronous operation (e.g. kernel)

• Get the error code for the last error:
  cudaError_t cudaMemcpyLastError(void)

• Get a string to describe the error:
  char *cudaGetErrorString(cudaError_t)

  printf("%s\n", cudaGetErrorString(cudaGetLastError()));
Device Management

- Application can query and select GPUs
  - `cudaGetDeviceCount(int *count)`
  - `cudaSetDevice(int device)`
  - `cudaGetDevice(int *device)`
  - `cudaGetDeviceProperties(cudaDeviceProp *prop, int device)`

- Multiple threads can share a device

- A single thread can manage multiple devices
  - `cudaSetDevice(i)` to select current device
  - `cudaMemcpy(...)` for peer-to-peer copies †
Introduction to CUDA C/C++

- What have we learned?
  - Write and launch CUDA C/C++ kernels
    - __global__, blockIdx.x, threadIdx.x, <<<>>>
  - Manage GPU memory
    - cudaMalloc(), cudaMemcpy(), cudaFree()
  - Manage communication and synchronization
    - __shared__, __syncthreads()
    - cudaMemcpy() VS cudaMemcpyAsync(), cudaDeviceSynchronize()
Compute Capability

• The **compute capability** of a device describes its architecture, e.g.
  – Number of registers
  – Sizes of memories
  – Features & capabilities

<table>
<thead>
<tr>
<th>Compute Capability</th>
<th>Selected Features</th>
<th>Tesla models</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>Fundamental CUDA support</td>
<td>870</td>
</tr>
<tr>
<td>1.3</td>
<td>Double precision, improved memory accesses, atomics</td>
<td>10-series</td>
</tr>
<tr>
<td>2.0</td>
<td>Caches, fused multiply-add, 3D grids, surfaces, ECC, P2P, concurrent kernels/copies, function pointers, recursion</td>
<td>20-series</td>
</tr>
</tbody>
</table>

• The following presentations concentrate on Kepler and Maxwell devices with Compute Capability >= 3.0
  – See [http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz4357A8KBg](http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz4357A8KBg) for details
### cudaGetDeviceProperties Output

<table>
<thead>
<tr>
<th></th>
<th>Tesla K10</th>
<th>Tesla K20</th>
<th>Tesla K40</th>
<th>Tesla K80</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Capability</td>
<td>3.0</td>
<td>3.5</td>
<td>3.5</td>
<td>3.7</td>
</tr>
<tr>
<td>globalL1CacheSupported</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>localL1CacheSupported</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
## L1 Cache/Shared Memory Configuration Options on KEPLER Compute 3.x Devices

<table>
<thead>
<tr>
<th>cudaFuncCachePreferShared</th>
<th>L1 Memory Size (KB)</th>
<th>Compute 3.0-3.5 Shared Memory Size (KB)</th>
<th>Compute 3.7 Shared Memory Size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cudaFuncCachePreferShared</td>
<td>16</td>
<td>48</td>
<td>112</td>
</tr>
<tr>
<td>cudaFuncCachePreferL1</td>
<td>48</td>
<td>16</td>
<td>80</td>
</tr>
<tr>
<td>cudaFuncCachePreferEqual</td>
<td>32</td>
<td>32</td>
<td>96</td>
</tr>
</tbody>
</table>
KEPLER SM (SMX)

- Scheduler not tied to cores
- Double issue for max utilization
MAXWELL SM (SMM)

SMM

- **Simplified design**
  - power-of-two, quadrant-based
  - scheduler tied to cores
- **Better utilization**
  - single issue sufficient
  - lower instruction latency
- **Efficiency**
  - $<10\%$ difference from SMX
  - $\sim50\%$ SMX chip area
Major enhancements

**KEPLER**

- CONTROL LOGIC

**MAXWELL 1st Generation**

- CONTROL LOGIC
- CONTROL LOGIC
- CONTROL LOGIC
- CONTROL LOGIC

135% Performance/Core

2x Performance/Watt
NEWS IN MAXWELL

- Functionally same as Kepler (K20)
- Dynamic parallelism is mainstream
- Architectural improvements
  - new SM design (SMM)
  - native shared-memory 32-bit atomics
  - other improvements
- CC 5.x (5.0 for GM 207, 5.2 for GM 204)
  - nvcc -arch=sm_50 ...
Shared-Memory Atomics

```c
__shared__ int l_n;
// . . .
atomicAdd(&l_n, 1);
```

pre-Maxwell (e.g. Kepler)

```
nvcc -arch=sm_35 ...
/*00c8*/ LDSLK P0, R2, [RZ];
/*00d0*/ @P0 IADD R2, R2, 0x1;
/*00d8*/ @P0 STSCUL P1, [RZ], R2;
/*00e0*/ @!P1 BRA 0xc0;
```

expensive lock-modify-unlock in a loop

Maxwell

```
nvcc -arch=sm_50 ...
/*00f0*/ @!P0 ATOMS.ADD RZ, [RZ], R4;
```

single instruction
Other Maxwell Features

- More thread blocks/SM
  - 32 (vs. 16), full occupancy with 64-thread TBs
- Shared memory
  - 64 KiB/SM in GM 207, 96 KiB/SM in GM204
  - 4-byte mode only
- L1 cache
  - read-only (same as texture cache)
  - not sharable with shared memory
- Larger L2 cache
  - 2 MiB vs. 1.5 MiB on Kepler
Histogram

Common operation in signal and image processing

on CPU:

```c
for(i = 0; i < n; i++)
    histo[data[i]]++;
```

on GPU:

```c
__global__ void histo_k
(int *histo, uchar *data, int n) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if(i >= n)
        return;
    atomicAdd(&histo[data[i]], 1);
}
```

Global atomics are the bottleneck
#define NCLASSES 256
#define BS 256
#define PER_THREAD 32
__global__ void histo_k(int *histo, const unsigned* data, int n) {
  // init per-block histogram
  __shared__ int lhisto[NCLASSES];
  for(int i = threadIdx.x; i < NCLASSES; i += BS)
    lhisto[i] = 0;
  __syncthreads();
  // compute per-block histogram
  int istart = blockIdx.x * (BS * PER_THREAD) + threadIdx.x;
  int iend = min(istart + BS * PER_THREAD, n);
  for(int i = istart; i < iend; i += BS) {
    union { unsigned char c[sizeof(unsigned)]; unsigned i; } el;
    el.i = data[i];
    for(int j = 0; j < sizeof(unsigned); j++)
      atomicAdd(&lhisto[el.c][j], 1); // shared-memory atomic
  }
  __syncthreads();
  // accumulate histogram to global storage
  for(int i = threadIdx.x; i < NCLASSES; i += BS)
    atomicAdd(&histo[i], lhisto[i]); // global atomics
} // histo_kernel
Histogram : Performance per SM

Higher performance expected with larger GPUs (more SMs)
### Scalability for the architecture: A summary of four generations

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Tesla</th>
<th>Fermi</th>
<th>Kepler</th>
<th>Maxwell</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>G80</td>
<td>GF100</td>
<td>GK104 (K10)</td>
<td>GK107 (GTX760)</td>
</tr>
<tr>
<td></td>
<td>GT200</td>
<td>GF104</td>
<td>GK110 (K20X)</td>
<td>GM204 (GTX980)</td>
</tr>
<tr>
<td></td>
<td>GF100</td>
<td>2010</td>
<td>GK110 (K40)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2011</td>
<td>2012</td>
<td>GK110 (K80)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2012</td>
<td>2013</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2013</td>
<td>2013</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2014</td>
<td>2014</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2014</td>
<td>2014</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2014</td>
<td>2014</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA Compute Capability</td>
<td>1.0</td>
<td>1.2</td>
<td>2.0</td>
<td>2.1</td>
<td>3.0</td>
<td>3.5</td>
<td>3.5</td>
<td>3.7</td>
<td>5.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N (multiproc.)</th>
<th>16</th>
<th>30</th>
<th>16</th>
<th>7</th>
<th>8</th>
<th>14</th>
<th>15</th>
<th>30</th>
<th>5</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>M (cores/multip.)</td>
<td>8</td>
<td>8</td>
<td>32</td>
<td>48</td>
<td>192</td>
<td>192</td>
<td>192</td>
<td>192</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>Number of cores</td>
<td>128</td>
<td>240</td>
<td>512</td>
<td>336</td>
<td>1536</td>
<td>2688</td>
<td>2880</td>
<td>5760</td>
<td>640</td>
<td>2048</td>
</tr>
</tbody>
</table>
INTRODUCING NCCL ("NICKEL"): ACCELERATED COLLECTIVES FOR MULTI-GPU SYSTEMS
INTRODUCING NCCL
Accelerating multi-GPU collective communications

GOAL:
• Build a research library of accelerated collectives that is easily integrated and topology-aware so as to improve the scalability of multi-GPU applications

APPROACH:
• Pattern the library after MPI’s collectives
• Handle the intra-node communication in an optimal way
• Provide the necessary functionality for MPI to build on top to handle inter-node
NCCL FEATURES AND FUTURES
(Green = Currently available)

- Broadcast
- All-Gather
- Reduce
- All-Reduce
- Reduce-Scatter
- Scatter
- Gather
- All-To-All
- Neighborhood

- Single-node, up to 8 GPUs
- Host-side API
- Asynchronous/non-blocking interface
- Multi-thread, multi-process support
- In-place and out-of-place operation
- Integration with MPI
- Topology Detection
- NVLink & PCIe/QPI* support
NCCL IMPLEMENTATION

Implemented as monolithic CUDA C++ kernels combining the following

• GPUDirect P2P Direct Access
• Three primitive operations: Copy, Reduce, ReduceAndCopy
• Intra-kernel synchronization between GPUs
• One CUDA thread block per ring-direction
#include <nccl.h>
ncclComm_t comm[4];
ncclCommInitAll(comm, 4, {0, 1, 2, 3});

foreach g in (GPUs) { // or foreach thread
    cudaSetDevice(g);
    double *d_send, *d_recv;
    // allocate d_send, d_recv; fill d_send with data
    ncclAllReduce(d_send,d_recv, N, ncclDouble, ncclSum, comm[g], stream[g]);
    // consume d_recv
}
NCCL PERFORMANCE

Bandwidth at different problem sizes (4 Maxwell GPUs)
AVAILABLE NOW

github.com/NVIDIA/nccl
Nsight Editor

Nsight Debugger

Nsight Profiler
UNIFIED PROFILING

- Analyze CPU threads + GPU kernels
- Identify performance bottlenecks
- Critical-path analysis

![Timeline Diagram]

CPU

A

B

wait

wait

5%

40%

Timeline

GPU

Kernel X

Kernel Y

Optimize Here
COMMON PROGRAMMING MODELS ACROSS MULTIPLE CPUS

Libraries
- AmgX
- cuDNN
- cuBLAS
- OpenCV
- Thrust

Compiler Directives
- OpenACC

Programming Languages
- C/C++
- Fortran
- Python
- Java

Platforms
- ARM
- x86
DEVELOP ON GEFORCE, DEPLOY ON TESLA

Designed for Developers & Gamers
Available Everywhere

Designed for the Data Center
- ECC
- 24x7 Runtime
- GPU Monitoring
- Cluster Management
- GPUDirect-RDMA
- Hyper-Q for MPI
- 3 Year Warranty
- Integrated OEM Systems, Professional Support
RESOURCES

Learn more about GPUs

CUDA resource center:
http://docs.nvidia.com/cuda

GTC on-demand and webinars:
http://on-demand-gtc.gputechconf.com
http://www.gputechconf.com/gtc-webinars

Parallel Forall Blog:
http://devblogs.nvidia.com/parallelforall

Self-paced labs:
http://nvidia.qwiklab.com
<table>
<thead>
<tr>
<th>KEY SPECS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>1 TFLOP/s 256-core Maxwell</td>
</tr>
<tr>
<td>CPU</td>
<td>64-bit ARM A57 CPUs</td>
</tr>
<tr>
<td>Memory</td>
<td>4 GB LPDDR4</td>
</tr>
<tr>
<td>Storage</td>
<td>16 GB eMMC</td>
</tr>
<tr>
<td>Wifi/BT</td>
<td>802.11 2x2 ac / BT Ready</td>
</tr>
<tr>
<td>Networking</td>
<td>1 Gigabit Ethernet</td>
</tr>
<tr>
<td>Size</td>
<td>50mm x 87mm</td>
</tr>
<tr>
<td>Interface</td>
<td>400 pin board-to-board connector</td>
</tr>
<tr>
<td>Power</td>
<td>Under 10W</td>
</tr>
</tbody>
</table>

**JETSON TX1**

Supercomputer on a module
JETSON LINUX SDK

Graphics

Deep Learning and Computer Vision

GPU Compute

Developer Tools

- OpenGL
- GLUT
- cuDNN
- CUDA

- NVIDIA VisionWorks
- NVDIA Tools extension

- Debugger
- Profiler
- System Trace
10X ENERGY EFFICIENCY FOR MACHINE LEARNING

Alexnet

![Graph showing energy efficiency comparison between Intel core i7-6700K (Skylake) and Jetson TX1. The graph indicates a 10x increase in efficiency for Jetson TX1 compared to Intel core i7-6700K.](image.png)
# Path to an Autonomous Drone

<table>
<thead>
<tr>
<th></th>
<th>Today’s Drone (GPS-Based)</th>
<th>Core i7</th>
<th>Jetson TX1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance*</td>
<td>1x</td>
<td>100x</td>
<td>100x</td>
</tr>
<tr>
<td>Power (compute)</td>
<td>2W</td>
<td>60W</td>
<td>6W</td>
</tr>
<tr>
<td>Power (mechanical)</td>
<td>70W</td>
<td>100W</td>
<td>80W</td>
</tr>
<tr>
<td>Flight Time</td>
<td>20 minutes</td>
<td>9 minutes</td>
<td>18 minutes</td>
</tr>
</tbody>
</table>

*Based on SGEMM performance*
Comprehensive developer platform

Jetson TX1 Developer Kit
$599 retail
$299 EDU
Pre-order Nov 12
Shipping Nov 16 (US)
Intl to follow
Jetson TX1 Module
$299 Available 1Q16
Distributors Worldwide
ONE ARCHITECTURE — END-TO-END AI

Tesla for Cloud
Titan X for PC
DRIVE PX for Auto
Jetson for Embedded
FIVE THINGS TO REMEMBER

Time of accelerators has come

NVIDIA is focused on co-design from top-to-bottom

Accelerators are surging in supercomputing

Machine learning is the next killer application for HPC

Tesla platform leads in every way