GPU Acceleration of Scientific Computing

jashley@nvidia.com
NVIDIA Corporation
GPU Acceleration
Accelerated Computing
GPU Accelerates Tasks Running on CPUs

CPU
Optimized for Serial Tasks

GPU Accelerator
Optimized for Many Parallel Tasks

10x Performance
5x Energy Efficiency
Low Latency or High Throughput?

**CPU**
- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution

**GPU**
- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation
Low Latency or High Throughput?

- **CPU architecture** must **minimize latency** within each thread.
- **GPU architecture** hides latency with computation from other thread warps.

![Diagram showing CPU and GPU architectures](image)
How GPU Acceleration Works

Application Code

Compute-Intensive Functions

Small slice of Code

Rest of Sequential CPU Code

GPU

CPU
Heterogeneous Computing

- **Terminology:**
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)
Biggest challenge facing supercomputers and scale out datacenters:

**POWER Efficiency**

GPU is purpose-built for power efficient computing for parallel applications, leading the way for the green datacenter.

Source: Intersect360 Research
University of Illinois scientists performed the **first all-atom simulation of the HIV virus** and discovered the chemical structure of its capsid —

“**the perfect target for fighting the infection.**”
Revolutionizing Scientific Computing

AMBER Molecular Dynamics Simulation
DHFR NVE Benchmark

64 Sandy Bridge CPUs
58 ns/day

1 Tesla K40 GPU
102 ns/day
US 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
CORAL: Built for Grand Scientific Challenges

**Fusion Energy**
Role of material disorder, statistics, and fluctuations in nanoscale materials and systems.

**Climate Change**
Study climate change adaptation and mitigation scenarios; realistically represent detailed features.

**Biofuels**
Search for renewable and more efficient energy sources.

**Astrophysics**
Radiation transport – critical to astrophysics, laser fusion, atmospheric dynamics, and medical imaging.

**Combustion**
Combustion simulations to enable the next gen diesel/biofuels to burn more efficiently.

**Nuclear Energy**
Unprecedented high-fidelity radiation transport calculations for nuclear energy applications.
Getting it the easy way
Research: Higher Education and Supercomputing

COMPUTATIONAL CHEMISTRY AND BIOLOGY

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>License</th>
<th>GPU Support</th>
<th>CPU Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIOCHEM</td>
<td>Sequence mapping software</td>
<td>OpenSource</td>
<td>1.0, 1.2, 2.0</td>
<td>Yes</td>
</tr>
<tr>
<td>CYSDyn</td>
<td>Density functional theory calculations</td>
<td>OpenSource</td>
<td>1.0, 1.2, 2.0</td>
<td>Yes</td>
</tr>
<tr>
<td>CYSDyn</td>
<td>Molecular dynamics simulations</td>
<td>OpenSource</td>
<td>1.0, 1.2, 2.0</td>
<td>Yes</td>
</tr>
</tbody>
</table>
| Gopi-BLAST | Large scale sequence alignment
 | OpenSource | 1.0, 1.2, 2.0 | Single only |
| Gopi-HMMER | Profile HMMer | OpenSource | 1.0, 1.2, 2.0 | Yes |

Molecular Dynamics

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>License</th>
<th>GPU Support</th>
<th>CPU Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alamod</td>
<td>Molecular dynamics of biological macromolecules, proteins, DNA and RNA</td>
<td>OpenSource</td>
<td>1.0, 1.2, 2.0</td>
<td>Single only</td>
</tr>
<tr>
<td>AMBER</td>
<td>Simulation of molecular mechanics and crystal thology</td>
<td>OpenSource</td>
<td>1.0, 1.2, 2.0</td>
<td>Yes</td>
</tr>
<tr>
<td>ANDES</td>
<td>Large scale simulations of RNA and DNA</td>
<td>OpenSource</td>
<td>1.0, 1.2, 2.0</td>
<td>Yes</td>
</tr>
</tbody>
</table>
| DR VME | Molecular dynamics package
 | OpenSource | 1.0, 1.2, 2.0 | Yes |

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>License</th>
<th>GPU Support</th>
<th>CPU Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAMPSS</td>
<td>Classical molecular dynamics package</td>
<td>OpenSource</td>
<td>1.0, 1.2, 2.0</td>
<td>Yes</td>
</tr>
<tr>
<td>LAMPSS</td>
<td>Simulations of biological systems</td>
<td>OpenSource</td>
<td>1.0, 1.2, 2.0</td>
<td>Yes</td>
</tr>
<tr>
<td>NAMD</td>
<td>Molecular dynamics simulations of proteins, DNA, RNA and RNA</td>
<td>OpenSource</td>
<td>1.0, 1.2, 2.0</td>
<td>Yes</td>
</tr>
</tbody>
</table>

290+ GPU-Accelerated Applications
www.nvidia.com/appscatalog
A bit more effort
3 Ways to Accelerate Applications

Applications

Libraries
“Drop-in” Acceleration

OpenACC Directives
Easily Accelerate Applications

Programming Languages
Maximum Flexibility
Easy, High-Quality Acceleration

- **Ease of use:** Using libraries enables GPU acceleration without in-depth knowledge of GPU programming

- **“Drop-in”:** Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes

- **Quality:** Libraries offer high-quality implementations of functions encountered in a broad range of applications

- **Performance:** NVIDIA libraries are tuned by experts
cuFFT Library

Features

- Single and double precision
- Real and complex data types
- Radix 2, 3, 5 and 7 natively supported
- 1D, 2D and 3D batched transforms

Interface

- Similar to the FFTW “Advanced Interface”
cuBLAS Library

Dense Linear Algebra
- Single and double precision
- Real and complex data types
- Vector- and matrix-vector operations
- Matrix-matrix operations

Interface
- Similar to Basic Linear Algebra Subprograms (BLAS)
- Supports dynamic parallelism (on K20)
- nvBLAS - Drop in support via library preload for standard BLAS
cuSPARSE Library

**Features**
- Format conversion (dense, CSR, block-CSR, ...)
- Sparse-dense (matrix-vector multiply and triangular solve)
- Sparse-sparse (matrix-matrix add and multiply)
- Preconditioners (incomplete-LU, tridiagonal, ...)

**Interface**
- C API with Fortran wrappers

![Different matrix sparsity patterns](image)
cuSOLVER

cusolverDN
- Dense Cholesky, LU, SVD, (batched) QR
- Optimization, Computer vision, CFD

cusolverSP
- Sparse direct solvers & Eigensolvers
- Newton’s method, Chemical kinetics

cusolverRF
- Sparse refactorization solver
- Chemistry, ODEs, Circuit simulation
cuRAND Library

Features
- Pseudo-RNGs* (XORWOW, MRG32k3a, MTGP)
- Quasi-RNGs (Sobol32, Sobol64)
- Uniform, Normal and Poisson distributions
- Statistical test results in documentation

Interface
- May be called from host routines and device kernels

*: Random Number Generators
Explore the CUDA (Libraries) Ecosystem

- CUDA Tools and Ecosystem described in detail on NVIDIA Developer Zone: developer.nvidia.com/cuda-tools-ecosystem

- Watch past GTC library talks
Descriptive Parallelism
3 Ways to Accelerate Applications

Applications

Libraries

“Drop-in” Acceleration

OpenACC Directives

Easily Accelerate Applications

Programming Languages

Maximum Flexibility
OpenACC Directives

Your original Fortran or C code

Program myscience
... serial code ...
!$acc kernels
do k = 1,n1
  do i = 1,n2
    ... parallel code ...
  enddo
enddo
!$acc end kernels
... End Program myscience

OpenACC Compiler Hint

CPU

GPU

Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs
Familiar to OpenMP Programmers

**OpenMP**

```c
main() {
    double pi = 0.0; long i;
    #pragma omp parallel for reduction(+:pi)
    for (i=0; i<N; i++)
    {
        double t = (double)((i+0.05)/N);
        pi += 4.0/(1.0+t*t);
    }
    printf("pi = %f\n", pi/N);
}
```

**OpenACC**

```c
main() {
    double pi = 0.0; long i;

    #pragma acc kernels
    for (i=0; i<N; i++)
    {
        double t = (double)((i+0.05)/N);
        pi += 4.0/(1.0+t*t);
    }
    printf("pi = %f\n", pi/N);
}
```
OpenACC Members and Supporters

- NVIDIA
- CRAY
- PGI
- CAPS
- Mentor
- Rogue Wave
- TOTAL
- OAK RIDGE National Laboratory
- NOAA
- CSCS
- EPCC
- University of Houston
- Indiana University
- Technische Universität Dresden
- Tokyo Tech
- Allinea
- LSU
- Sandia National Laboratories
Directives: Easy & Powerful

Real-Time Object Detection
Global Manufacturer of Navigation Systems

Valuation of Stock Portfolios using Monte Carlo
Global Technology Consulting Company

Interaction of Solvents and Biomolecules
University of Texas at San Antonio

5x in 40 Hours  2x in 4 Hours  5x in 8 Hours

“Optimizing code with directives is quite easy, especially compared to CPU threads or writing CUDA kernels. The most important thing is avoiding restructuring of existing code for production applications.”

--- Developer at the Global Manufacturer of Navigation Systems
Focus on Exposing Parallelism

With Directives, tuning work focuses on exposing parallelism, which makes codes inherently better.

Example: Application tuning work using directives for new Titan system at ORNL

S3D
Research more efficient combustion with next-generation fuels

- Tuning top 3 kernels (90% of runtime)
- 3 to 6x faster on CPU+GPU vs. CPU+CPU
- But also improved all-CPU version by 50%

CAM-SE
Answer questions about specific climate change adaptation and mitigation scenarios

- Tuning top key kernel (50% of runtime)
- 6.5x faster on CPU+GPU vs. CPU+CPU
- Improved performance of CPU version by 100%
A Very Simple Exercise: SAXPY

**SAXPY in C**

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

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

**SAXPY in Fortran**

```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)
```

A Very Simple Exercise: SAXPY
Directive Syntax

- **Fortran**
  
  ```fortran
  !$acc directive [clause [,] clause] ...
  ```

  Often paired with a matching end directive surrounding a structured code block
  
  ```fortran
  !$acc end directive
  ```

- **C**
  
  ```c
  #pragma acc directive [clause [,] clause] ...
  ```

  Often followed by a structured code block
kernels: Your first OpenACC Directive

Each loop executed as a separate *kernel* on the GPU.

```c
!$acc kernels
  do i=1,n
    a(i) = 0.0
    b(i) = 1.0
    c(i) = 2.0
  end do

  do i=1,n
    a(i) = b(i) + c(i)
  end do

!$acc end kernels
```

**Kernel:** A parallel function that runs on the GPU
Kernels Construct

Fortran

```fortran
 !$acc kernels [clause ...]
   structured block
 !$acc end kernels
```

C

```c
#pragma acc kernels [clause ...]
 { structured block }
```

Clauses

```c
if( condition )
async( expression )
```

Also, any data clause (more later)
Complete SAXPY example code

Trivial first example
- Apply a loop directive
- Learn compiler commands

```c
#include <stdlib.h>

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

int main(int argc, char **argv)
{
    int N = 1<<20;  // 1 million floats
    if (argc > 1)
        N = atoi(argv[1]);

    float *x = (float*)malloc(N * sizeof(float));
    float *y = (float*)malloc(N * sizeof(float));

    for (int i = 0; i < N; ++i)
    {
        x[i] = 2.0f;
        y[i] = 1.0f;
    }

    saxpy(N, 3.0f, x, y);

    return 0;
}
```

*restrict: “I promise y does not alias x”
Compile and run

- **C**: `pgcc -acc -ta=nvidia -Minfo=accel -o saxpy_acc saxpy.c`
- **Fortran**: `pgf90 -acc -ta=nvidia -Minfo=accel -o saxpy_acc saxpy.f90`
- **Compiler output:**

```
pgcc -acc -Minfo=accel -ta=nvidia -o saxpy_acc saxpy.c  
saxpy:  
  8, Generating copyin(x[:n-1])  
    Generating copy(y[:n-1])  
    Generating compute capability 1.0 binary  
    Generating compute capability 2.0 binary  
  9, Loop is parallelizable  
    Accelerator kernel generated  
    9, #pragma acc loop worker, vector(256) /* blockIdx.x threadIdx.x */  
    CC 1.0 : 4 registers; 52 shared, 4 constant, 0 local memory bytes; 100% occupancy  
    CC 2.0 : 8 registers; 4 shared, 64 constant, 0 local memory bytes; 100% occupancy
```
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: $\nabla^2 f(x, y) = 0$

$$A_{k+1}(i, j) = \frac{A_k(i-1, j) + A_k(i+1, j) + A_k(i, j-1) + A_k(i, j+1)}{4}$$
while ( error > tol && iter < iter_max ) {
    error=0.0;

    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++) {

            error = max(error, 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 Fortran Code

```fortran
    do while ( err > tol .and. iter < iter_max )
      err=0._fp_kind
      do j=1,m
        do i=1,n
          Anew(i,j) = .25._fp_kind * (A(i+1, j ) + A(i-1, j ) + &
                        A(i , j-1) + A(i , j+1))
          err = max(err, Anew(i,j) - A(i,j))
        end do
      end do
      do j=1,m-2
        do i=1,n-2
          A(i,j) = Anew(i,j)
        end do
      end do
      iter = iter +1
    end do
```
Exercise 1

Jacobi kernels

- Task: use acc kernels to parallelize the Jacobi loop nests
Exercise 1 Solution: OpenACC C

while ( error > tol && iter < iter_max ) {
    error=0.0;
    
    #pragma acc kernels
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++ ) {
            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }
    
    #pragma acc kernels
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
**Exercise 1 Solution: OpenACC Fortran**

```fortran
do while ( error > tol .and. iter < iter_max )
err=0._fp_kind

$!acc kernels
  do j=1,m
    do i=1,n
      Anew(i,j) = 0.25 * (A(i+1,j) + A(i-1,j) + &
                        A(i,j-1) + A(i,j+1))
      err = max(err, abs(Anew(i,j) - A(i,j));
    enddo
  enddo
$!acc end kernels

$!acc kernels
  do j=1, m-2
    do i=1, n-2
      A(i,j) = Anew(i,j)
    enddo
  enddo
$!acc end kernels
iter = iter+1
enddo
```

Execute GPU kernel for loop nest

Execute GPU kernel for loop nest
Exercise 1: Compiler output (C)

```
pgcc -tp sandybridge-64 -acc -ta=nvidia -Minfo=accel -o laplace2d_acc laplace2d.c
main:
  56, Generating present_or_copyout(Anew[1:4094][1:4094])
  Generating present_or_copyin(A[:][:])
  Generating Tesla code
  57, Loop is parallelizable
  59, Loop is parallelizable
  Accelerator kernel generated
  57, #pragma acc loop gang /* blockIdx.y */
  59, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
  63, Max reduction generated for error
  68, Generating present_or_copyin(Anew[1:4094][1:4094])
  Generating present_or_copyout(A[1:4094][1:4094])
  Generating Tesla code
  69, Loop is parallelizable
  71, Loop is parallelizable
  Accelerator kernel generated
  69, #pragma acc loop gang /* blockIdx.y */
  71, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```
## Exercise 1: Performance

**CPU:** Intel E5-2670
8 Cores @ 2.60 GHz

**GPU:** NVIDIA Tesla K520m

<table>
<thead>
<tr>
<th>Execution (4096x4096)</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 1 OpenMP thread</td>
<td>81.2</td>
<td>--</td>
</tr>
<tr>
<td>CPU 2 OpenMP threads</td>
<td>42.4</td>
<td>1.91x</td>
</tr>
<tr>
<td>CPU 4 OpenMP threads</td>
<td>25.7</td>
<td>3.16x</td>
</tr>
<tr>
<td>CPU 8 OpenMP threads</td>
<td>21.6</td>
<td>5.35x</td>
</tr>
<tr>
<td>OpenACC GPU</td>
<td>148.5</td>
<td>0.15x FAIL</td>
</tr>
</tbody>
</table>

**Speedup vs. 1 CPU core**

**Speedup vs. 8 CPU cores**
What went wrong?

**PGI_ACC_TIME=1**

time(us): 8,287,436

56: data region reached 1000 times
   56: data copy in transfers: 8000
   device time(us): total=182,302 max=52 min=7 avg=22
   68: data copy out transfers: 8000
   device time(us): total=226,025 max=118 min=7 avg=28

56: compute region reached 1000 times
59: kernel launched 1000 times
   grid: [32x4094]  block: [128]
   device time(us): total=5,000,928 max=5,004 min=4,999
   avg=5,000

elapsed time(us): total=5,045,006 max=6,686 min=5,038
avg=5,045

59: reduction kernel launched 1000 times
   grid: [1]  block: [256]
   device time(us): total=239,763 max=241 min=239 avg=239
   elapsed time(us): total=262,790 max=332 min=259 avg=262

68: data region reached 1000 times

Huge Data Transfer Bottleneck!
Computation: 10 seconds
Data malloc/movement: 138
For efficiency, decouple data movement and compute off-load.
Excessive Data Transfers

while (error > tol && iter < iter_max) {
    error = 0.0;
    #pragma acc kernels
    for (int j = 1; j < n-1; j++) {
        for (int i = 1; i < m-1; i++) {
            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }
    A, Anew resident on host
    Copy
    A, Anew resident on accelerator
    These copies happen every iteration of the outer while loop!*

A, Anew resident on host
Copy
A, Anew resident on accelerator

A, Anew resident on host
Copy
A, Anew resident on accelerator

*Note: there are two #pragma acc kernels, so there are 4 copies per while loop iteration!
Data Management
Data Construct

Fortran

 !$acc data [clause …]
     structured block
 !$acc end data

C

 #pragma acc data [clause …]
     { structured block }

General Clauses

   if( condition )
   async( expression )

Manage data movement. Data regions may be nested.
Data Clauses

- **copy** (list) Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.
- **copyin** (list) Allocates memory on GPU and copies data from host to GPU when entering region.
- **copyout** (list) Allocates memory on GPU and copies data to the host when exiting region.
- **create** (list) Allocates memory on GPU but does not copy.
- **present** (list) Data is already present on GPU from another containing data region.

And **present_or_copy**[in|out], **present_or_create**, **deviceptr**.
Exercise 2: Jacobi Data Directives

- Task: use **acc data** to minimize transfers in the Jacobi example
#pragma acc data copy(A), create(Anew)
while ( error > tol && iter < iter_max ) {
    error=0.0;

    #pragma acc kernels
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++ ) {
            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }

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

    iter++;
}
Exercise 2 Solution: OpenACC Fortran

```fortran
!$acc data copy(A), create(Anew)
do while ( err > tol .and. iter < iter_max )
   err=0._fp_kind

!$acc kernels
   do j=1,m
      do i=1,n

         Anew(i,j) = .25 fp_kind * (A(i+1, j ) + A(i-1, j ) + &
                                  A(i , j-1) + A(i , j+1))

         err = max(err, Anew(i,j) - A(i,j))
      end do
   end do
!$acc end kernels

...  

iter = iter +1
end do
!$acc end data
```
## Exercise 2: Performance

<table>
<thead>
<tr>
<th>Execution (4096x4096)</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 8 OpenMP thread</td>
<td>21.6</td>
<td>--</td>
</tr>
<tr>
<td>OpenACC K520m</td>
<td>148.5</td>
<td>0.15x</td>
</tr>
<tr>
<td>OpenACC K520m-opt</td>
<td>7.75</td>
<td>2.79x</td>
</tr>
</tbody>
</table>
What went right?

time(us): 7,423,576
50: data region reached 1 time
50: data copyin transfers: 16
  device time(us): total=305 max=35 min=8 avg=19
82: data copyout transfers: 18
  device time(us): total=182 max=28 min=1 avg=10
56: compute region reached 1000 times
59: kernel launched 1000 times
  grid: [32x4094] block: [128]
  device time(us): total=5,000,403 max=5,003 min=4,998 avg=5,000
  elapsed time(us): total=5,026,969 max=5,194 min=5,020 avg=5,026
59: reduction kernel launched 1000 times
  grid: [1] block: [256]
  device time(us): total=239,865 max=239,912 min=239,823 avg=239
  elapsed time(us): total=261,417 max=502 min=258 avg=261
Transfer Bottleneck Eliminated!
Computation: 10 seconds
Data movement: negligible
Further speedups

- OpenACC gives us more detailed control over parallelization
  - Via gang, worker, and vector clauses
- By understanding more about OpenACC execution model and GPU hardware organization, we can get higher speedups on this code
- By understanding bottlenecks in the code via profiling, we can reorganize the code for higher performance
Finding Parallelism in your code

- (Nested) for loops are best for parallelization
- Large loop counts needed to offset GPU/memcpy overhead
- Iterations of loops must be independent of each other
  - To help compiler: restrict keyword (C), independent clause
- Compiler must be able to figure out sizes of data regions
  - Can use directives to explicitly control sizes
- Pointer arithmetic should be avoided if possible
  - Use subscripted arrays, rather than pointer-indexed arrays.
For the control freaks
3 Ways to Accelerate Applications

Applications

Libraries
“Drop-in” Acceleration

OpenACC Directives
Easily Accelerate Applications

Programming Languages
Maximum Flexibility
Disclaimer on the CUDA programming model

- I will present the basic rules of the programming model.

- In almost all cases, there are advanced features of the model that let you bend the rules ... but that is a different and much longer session!
CUDA C++ Template Library

- Optimized parallel algorithms, include
  - Scan
  - Sort
  - Transform
  - Reduce

- Interface
  - Host and device containers that mimic the C++ STL
  - Allows quick prototyping
  - OpenMP backend for portability

Also available on: http://thrust.github.com
# Code Example

```cpp
#include <thrust/scan.h>

int data[6] = {1, 0, 2, 2, 1, 3};

thrust::inclusive_scan(data, data + 6, data); // in-place scan

// data is now {1, 1, 3, 5, 6, 9}
```

Also available on: http://thrust.github.com
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);

www.boost.org/libs/lambda

---

int N = 1<<20;
thrust::host_vector<float> x(N), y(N);
...

thrust::device_vector<float> d_x = x;
thrust::device_vector<float> d_y = y;

// Perform SAXPY on 1M elements
thrust::transform(d_x.begin(), d_x.end(),
d_y.begin(), d_y.begin(),
2.0f * _1 + _2);

thrust.github.com
Thrust Performance

- Thrust 5.0 on K20X, input and output data on device
- TBB 4.1 on Intel SandyBridge E5-2687W @3.10GHz

Performance may vary based on OS version and motherboard configuration
Anatomy of a CUDA Application

- **Serial** code executes in a **Host** (CPU) thread
- **Parallel** code executes in many **Device** (GPU) threads across multiple processing elements
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
Basic GPU Layout (Fermi & Kepler)
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
Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
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
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
Kernel Execution

- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time

CUDA thread

CUDA thread block

CUDA kernel grid

CUDA core

CUDA Streaming Multiprocessor

CUDA-enabled GPU

- Each thread is executed by a core
- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks’ memory requirements and the SM’s memory resources
- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time
A kernel is launched as a grid of blocks of threads

**Built-in variables:**
- threadIdx
- blockIdx
- blockDim
- gridDim
Thread blocks allow cooperation

Threads may need to cooperate:

- Cooperatively load/store blocks of memory that they all use
- Share results with each other or cooperate to produce a single result
- Synchronize with each other
Thread blocks allow scalability

- Blocks can execute in any order, concurrently or sequentially
- This independence between blocks gives scalability:
  - A kernel scales across any number of SMs
A thread block consists of 32-thread warps.

A warp is executed physically in parallel (SIMD) on a multiprocessor.
Warps

- Blocks are divided into 32 thread wide units called warps
  - Size of warps is implementation specific and can change in the future

- The SM creates, manages, schedules and executes threads at warp granularity
  - Each warp consists of 32 threads of contiguous threadIds

- All threads in a warp execute the same instruction
  - If threads of a warp diverge the warp serially executes each branch path taken

- When a warp executes an instruction that accesses global memory it coalesces the memory accesses of the threads within the warp into as few transactions as possible
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 stencil_1d() 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;
}
```

parallel fn

serial code

parallel code

serial code
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 memory**
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`
Multiple Memory Spaces

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

- 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
Memory hierarchy

- **Thread:**
  - Registers
  - Local memory

- **Block of threads:**
  - Shared memory

- **All blocks:**
  - Global memory

Global memory
Typical lines of CUDA C++ code on the **HOST**

//allocate device memory
cudaMalloc((void**)&device_ptr, how_many*sizeof(double));

//copy data between host <-> device
cudaMemcpy(target_ptr, destination_ptr, how_many*sizeof(double), cudaMemcpyDeviceToHost);

//GPU Accelerated Function Launch
someFunction<<<numBlocks,numThreads>>>( ...function parameters... ) ;

//Library call
curandGenerateNormal(gen_handle,device_ptr,how_many,mean,stddev);
**Typical CUDA C++ code on the DEVICE**

```c
__global__ void dFoo( double *bar, ... , int lenOfBar) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x ;
    int stride = gridSize.x * blockDim.x ;

    for( idx; idx < lenOfBar; idx += stride ) { // loop over all
        ...
        do some C++ stuff to bar[idx] here
        ...
    } // end for
} // end dFoo
```

CUDA C++ code is written from the perspective of a single thread.
Example 1
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);
}

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];
    }
}
```
IDs and Dimensions

- **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

| Thread | 0 | 1 | 2 | ... | 255 | 256 | ... | 511 | 512 | ... | 767 | ... | N |
|--------|---|---|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|---|
Parallelization of VecAdd

<table>
<thead>
<tr>
<th>Thread</th>
<th>threadBlock 0</th>
<th>threadBlock 1</th>
<th>threadBlock 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>255</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>256</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>511</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>512</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>767</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Thread inputs and outputs.
Parallelization of VecAdd

```
work index i = threadIdx.x + blockIdx.x * blockDim.x;
```

index of the thread within a thread block
index of the threadblock within the grid
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.

```c
#include <cuda_runtime.h>

__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];
    }
}
```

i is a different value for each thread.

Protect against invalid access if too many threads are launched.
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);
    cudaMemcpy(devC, c, N*sizeof(float), cudaMemcpyHostToDevice);
}
Step 4: Launch the GPU Kernel

```c
void main()
{
  ...
  vecAdd<<<(N+127)/128, 128>>>(N, devA, devB, devC);
  ...
}
```

- Call function by name as usual.
- Angle Brackets: Specify `launch configuration` for the kernel.
- 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`.
- Normal parameter passing syntax. Note that `devA`, `devB`, and `devC` are `device pointers`. They point to memory allocated on the GPU.
Step 5: Copy data back.

```c
void main()
{
    ...

    cudaMemcpy(c, devC, N*sizeof(float), cudaMemcpyDeviceToHost);

    ...
}
```
Example 2
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 \texttt{\_\_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

![Diagram showing halo and blockDim.x output elements]
__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];
  }
}
// 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];
if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
int result = 0;
result += temp[lindex + 1];
```

- **Store at temp[18]**
- **Skipped, threadIdx > RADIUS**
- **Load from temp[19]**
__syncthreads()

```c
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) {
    __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];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();
}
// Apply the stencil
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

// Store the result
out[gindex] = result;
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()` is 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 (\texttt{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:
\texttt{cudaError_t \texttt{cudaGetLastError}(void)}

Get a string to describe the error:
\texttt{char *\texttt{cudaGetErrorString}(cudaError_t)}

\texttt{printf("%s\n", \texttt{cudaGetErrorString(\texttt{cudaGetLastError()}));}
GPU Roadmap

- 2008: Tesla, CUDA
- 2010: Fermi, FP64
- 2012: Kepler, Dynamic Parallelism
- 2014: Maxwell, DX12
- 2016: Pascal
  - Unified Memory
  - 3D Memory
  - NVLink
  - Pascal
  - Unified Memory
  - 3D Memory
  - NVLink
Pascal GPU Features
NVLINK and Stacked Memory

**NVLINK**
- GPU high speed interconnect
- 80-200 GB/s

**3D Stacked Memory**
- 4x Higher Bandwidth (~1 TB/s)
- 3x Larger Capacity
- 4x More Energy Efficient per bit
NVLink
High-Speed GPU Interconnect

KEPLER GPU

PASCAL GPU

X86, ARM64, POWER CPU

2014

X86, ARM64, POWER CPU

2016

POWER CPU

NVLink

PCIe
Questions?

Now you know CUDA C/C++. Go forth and accelerate!