INTRODUCTION TO NVIDIA GPU COMPUTING

CADES Town Hall, November 2019
Jeff Larkin (jlarkin@nvidia.com)
Robert Searles (rsearles@nvidia.com)
AGENDA

Introduction to NVIDIA GPUs

GPU Computing Fundamentals
What are GPUs? Why and how should I use them?

Introduction to OpenACC
The simplest way to get started.

For More Information
INTRODUCTION TO NVIDIA GPUS
THE WORLD LEADER IN VISUAL COMPUTING
Tesla Accelerates Discoveries

Using a supercomputer powered by the Tesla Platform with over 3,000 Tesla accelerators, 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."

Without GPUs, the supercomputer would need to be 5x larger for similar performance.
TWO FORCES SHAPING COMPUTING

For 30 years, the dynamics of Moore’s law held true. But now CPU scaling is slowing while the demand for computing power surges ahead.

With AI, machines can learn. AI can solve grand challenges that have been beyond human reach. But it must be fueled by massive compute power.

Accelerated computing is the path forward beyond Moore’s law, delivering 1,000X computing performance every 10 years.
ONE ARCHITECTURE

NVIDIA is an accelerated computing company. It starts with a highly specialized parallel processor called the GPU and continues through system design, system software, algorithms, and optimized applications.

We leverage a single architecture across our growth markets — from gaming to transportation to healthcare — that is supported by 1.2 million developers today.
POWERING THE WORLD’S FASTEST SUPERCOMPUTERS

GPU acceleration is the most accessible and energy-efficient path forward for the world’s most powerful computers. More than 600 applications support CUDA today, including the top 15 in HPC.

NVIDIA powers U.S.-based Summit, the world’s fastest supercomputer, as well as the fastest systems in Europe and Japan. 27,000 NVIDIA Volta Tensor Core GPUs accelerate Summit’s performance to more than 200 petaflops for HPC and 3 exaops for AI.
GPU COMPUTING FUNDAMENTALS
Accelerated Computing

10x Performance & 5x Energy Efficiency for HPC

CPU
Optimized for Serial Tasks

GPU Accelerator
Optimized for Parallel Tasks
Accelerated Computing
10x Performance & 5x Energy Efficiency for HPC

CPU
Optimized for Serial Tasks

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

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

GPU Accelerator
Optimized for Parallel Tasks

CPU Strengths
Accelerated Computing

10x Performance & 5x Energy Efficiency for HPC

GPU Strengths

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

GPU Weaknesses

• Relatively low memory capacity
• Low per-thread performance
Speed v. Throughput

Speed

Throughput

Which is better depends on your needs...

*Images from Wikimedia Commons via Creative Commons
Amdahl’s law is an observation that how much speed-up you get from parallelizing the code is limited by the remaining serial part.

Any remaining serial code will reduce the possible speed-up.

This is why it’s important to focus on parallelizing more of your code before optimizing individual parts.
What’s the maximum speed-up that can be obtained by parallelizing 50% of the code?

\[
( \frac{1}{100\% \text{-} 50\%} ) = (1 / 1.0 - 0.50 ) = 2.0X
\]

What’s the maximum speed-up that can be obtained by parallelizing 25% of the code?

\[
( \frac{1}{100\% \text{-} 25\%} ) = (1 / 1.0 - 0.25 ) = 1.3X
\]

What’s the maximum speed-up that can be obtained by parallelizing 90% of the code?

\[
( \frac{1}{100\% \text{-} 90\%} ) = (1 / 1.0 - 0.90 ) = 10.0X
\]
What does Amdahl’s Law teach Us?

It is critical to understand the profile of your code to predict possible speed-ups.

Even a small amount of serialization can negatively affect performance.

Prioritize the most time-consuming routines and any code that forces serialization.
3 Ways to Accelerate Applications

- **Applications**
  - Libraries
    - Easy to use
    - Most Performance
  - Compiler Directives
    - Easy to use
    - Portable code
  - Programming Languages
    - Most Performance
    - Most Flexibility
CUDA TOOLKIT
Libraries, Languages and Development Tools for GPU Computing

**Programming Approaches**
- Libraries: "Drop-in" Acceleration
- Compiler Directives: Ease of use
- Programming Languages: Maximum Flexibility

**Development Environment**
- Nsight Systems
- Nsight Compute
- CUDA Profiling Tools Interface
- CUDA-GDB Debugger
- CUDA MEMCHECK

**Language Support**
- C
- C++
- Fortran
- python

Compile new languages to CUDA
3 Ways to Accelerate Applications

Applications

Libraries
- Easy to use
- Most Performance

Compiler Directives
- Easy to use
- Portable code

Programming Languages
- Most Performance
- Most Flexibility
LIBRARIES: 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
Math and Communication

cuBLAS

cuSPARSE

cuTENSOR

cuSOLVER

cuRAND

cuFFT

CUDA Math API

NVSHMEM
Major Initiatives

- **Performance**
  - Tuning & new algorithms

- **Extended Features**
  - New libraries & APIs

- **Multi-GPU**
  - Strong & weak scaling

- **Single GPU**
  - Tensor Cores

---

### NVIDIA HPC Libraries

#### Major Initiatives

- **Extended Features**
  - New libraries & APIs

- **Multi-GPU**
  - Strong & weak scaling

- **Single GPU**
  - Tensor Cores

---

**cuFFT Performance on DGX-2 for 3D C2C Single Precision FFTs**

- cuFFTDX
- cuTENSOR
- NVSHMEM

---

**Graph**

- 1024 x 1024 x 1024
- 512 x 512 x 512
cuBLAS

GPU-accelerated library for dense linear algebra

Accelerated library with complete BLAS plus extensions

- Supports all 152 standard routines for single, double, complex, and double complex
- Supports half-precision (FP16), integer (INT8) matrix and mixed precision multiplication operations
- Batched routines for higher performance on small problem sizes
- Host and device-callable interface
- XT interface supports distributed computations across multiple GPUs

https://developer.nvidia.com/cublas
cuSPARSE
Sparse Linear Algebra on GPUs

Optimized Sparse Matrix Library

- Optimized sparse linear algebra BLAS routines for matrix-vector, matrix-matrix, triangular solve
- Support for variety of formats (CSR, COO, block variants)
- Incomplete-LU and Cholesky preconditioners
- Support for half-precision (fp16) sparse matrix-vector operations

https://developer.nvidia.com/cusparse
cuSOLVER
Linear Solver Library

Library for Dense and Sparse Direct Solvers

Supports Dense Cholesky, LU, (batched) QR, SVD and Eigenvalue solvers

Sparse direct solvers & Eigen solvers

Includes a sparse refactorization solver for solving sequences of matrices with a shared sparsity pattern

Used in a variety of applications such as circuit simulation and computational fluid dynamics

Sample Applications
• Computer Vision
• CFD
• Newton’s method
• Chemical Kinetics
• Chemistry
• ODEs
• Circuit Simulation

https://developer.nvidia.com/cusolver
Tensor Core Accelerated Linear Solvers

Mixed-precision Dense Linear Solvers in cuSOLVER

- **LU Solver**
  - Q4 2019
  - Real & Complex FP32 & FP64
- **Cholesky Solver**
  - Coming Soon
  - Real & Complex FP32 & FP64
- **QR Solver**
  - Coming Soon
  - Real & Complex FP32 & FP64

- Solve dense linear system by one-sided factorizations
- Retain FP64 accuracy with ~3X Tensor Core Acceleration
cuTENSOR

A New High Performance CUDA Library for Tensor Primitives

Now
- Tensor Contractions and Reductions
- Elementwise Operations
- Mixed Precision Support
- Elementwise Fusion
- Coming Soon
  - HMMA Acceleration
- Available in Math Library EA Program

---

cuTENSOR vs TBLIS
1000 Random 3D-6D Contractions

developer.nvidia.com/CUDAMathLibraryEA
MULTI GPU MATH LIBRARIES

**cuFFT**

Current Support
- Up to 16 GPUs in a single process
- Single and double precision
- 1D C2C
- 2D/3D C2C, R2C, and C2R

**cuSOLVERMg**

CUDA Toolkit 10.1 Update 2
- Single Process Multi GPU Symmetric Eigensolver
- Best in class performance

CUDA Toolkit 10.2 (SC’19)
- Single Process Multi GPU LU

**cuBLASMg**

Available in CUDA Math Library EA Program
- Single Process Multi GPU GEMM
- State of the art, asymptotically peak performance

---

### cuFFT Performance on DGX-2 for 3D C2C Single Precision FFTs

- *1024 x 1024 x 1024*
- *512 x 512 x 512*

### PDSYEV performance for 20k Matrix on DGX-2 Tesla V100

- *cuSOLVERMg*  
- *MAGMA*

### FP32 SPMG GEMM Performance

- *cuBLASMg 4 V100*
- *cuBLASMg 8 V100*
- *4 V100 Peak*
- *8 V100 Peak*
cuFFT
Complete Fast Fourier Transforms Library

Complete Multi-Dimensional FFT Library

“Drop-in” replacement for CPU FFTW library
Real and complex, single- and double-precision data types
Includes 1D, 2D and 3D batched transforms
Support for half-precision (FP16) data types
Supports flexible input and output data layouts
XT interface now supports up to 8 GPUs

https://developer.nvidia.com/cufft
3 Ways to Accelerate Applications

- **Libraries**: Easy to use, Most Performance
- **Compiler Directives**: Easy to use, Portable code
- **Programming Languages**: Most Performance, Most Flexibility
Programming Languages for GPU Computing

SIX WAYS TO SAXPY
SINGLE PRECISION ALPHA X PLUS Y (SAXPY)

Part of Basic Linear Algebra Subroutines (BLAS) Library

\[ z = \alpha x + y \]

\( x, y, z : \text{vector} \)
\( \alpha : \text{scalar} \)

GPU SAXPY in multiple languages and libraries

A menagerie\* of possibilities, not a tutorial

\*technically, a program chrestomathy: http://en.wikipedia.org/wiki/Chrestomathy
Serial BLAS Code

```c
int N = 1<<20;
...

// Use your choice of blas library

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

Parallel cuBLAS Code

```c
int N = 1<<20;

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

// Perform SAXPY on 1M elements
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);
cublasGetVector(N, sizeof(y[0]), d_y, 1, y, 1);
cublasShutdown();
```

You can also call cuBLAS from Fortran, C++, Python, and other languages

http://developer.nvidia.com/cublas
OpenACC COMPILER DIRECTIVES

Parallel C Code

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

Parallel Fortran Code

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

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

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

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

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

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

int N = 1<<20;
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));

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

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

http://thrust.github.com
CUDA FORTRAN

**Standard Fortran**

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

program main
  use mymodule
  real :: x(2**20), y(2**20)
  x = 1.0, y = 2.0
  ! Perform SAXPY on 1M elements
  call saxpy(2**20, 2.0, x, y)
end program main

**Parallel Fortran**

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

program main
  use cudafor; use mymodule
  real, device :: x_d(2**20), y_d(2**20)
  x_d = 1.0, y_d = 2.0
  ! Perform SAXPY on 1M elements
  call saxpy<<<4096,256>>>(2**20, 2.0, x_d, y_d)
end program main

import numpy as np

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

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

cpu_result = saxpy(2.0, x, y)

import numpy as np
from numba import vectorize

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

N = 1048576

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

# Add arrays on GPU
C = saxpy(2.0, X, Y)
CUDA TRAINING SERIES COMING IN JANUARY!
INTRODUCTION TO OPENACC
OpenACC is a directives-based programming approach to parallel computing designed for performance and portability on CPUs and GPUs for HPC.

```c
main()
{
    <serial code>
    #pragma acc kernels
    {
        <parallel code>
    }
}
```
3 WAYS TO ACCELERATE APPLICATIONS

- Libraries: Easy to use, Most Performance
- Compiler Directives: Easy to use, Portable code
- Programming Languages: Most Performance, Most Flexibility

OpenACC
OpenACC is designed to be portable to many existing and future parallel platforms.

The programmer need not think about specific hardware details, but rather express the parallelism in generic terms.

An OpenACC program runs on a *host* (typically a CPU) that manages one or more parallel *devices* (GPUs, etc.). The host and device(s) are logically thought of as having separate memories.
OPENACC

**Incremental**
- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate more of the code

**Single Source**
- Rebuild the same code on multiple architectures
- Compiler determines how to parallelize for the desired machine
- Sequential code is maintained

**Low Learning Curve**
- OpenACC is meant to be easy to use, and easy to learn
- Programmer remains in familiar C, C++, or Fortran
- No reason to learn low-level details of the hardware.
OPENACC SUCCESSES

**LSDalton**
- Quantum Chemistry
- Aarhus University
- 12X speedup
- 1 week

**PowerGrid**
- Medical Imaging
- University of Illinois
- 40 days to 2 hours

**COSMO**
- Weather and Climate
- MeteoSwiss, CSCS
- 40X speedup
- 3X energy efficiency

**INCOMP3D**
- CFD
- NC State University
- 4X speedup

**NekCEM**
- Comp Electromagnetics
- Argonne National Lab
- 2.5X speedup
- 60% less energy

**MAESTRO CASTRO**
- Astrophysics
- Stony Brook University
- 4.4X speedup
- 4 weeks effort

**CloverLeaf**
- Comp Hydrodynamics
- AWE
- 4X speedup
- Single CPU/GPU code

**FINE/Turbo**
- CFD
- NUMECA International
- 10X faster routines
- 2X faster app
OPENACC SYNTAX
A **pragma** in C/C++ gives instructions to the compiler on how to compile the code. Compilers that do not understand a particular pragma can freely ignore it.

A **directive** in Fortran is a specially formatted comment that likewise instructions the compiler in it compilation of the code and can be freely ignored.

“**acc**” informs the compiler that what will come is an OpenACC directive.

**Directives** are commands in OpenACC for altering our code.

**Clauses** are specifiers or additions to directives.
EXAMPLE CODE
We will observe a simple simulation of heat distributing across a metal plate.

We will apply a consistent heat to the top of the plate.

Then, we will simulate the heat distributing across the plate.
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 ( err > tol && iter < iter_max ) {
    err=0.0;

    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]));
        }
    }

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

    iter++;
}
PROFILE-DRIVEN DEVELOPMENT
OPENACC DEVELOPMENT CYCLE

- **Analyze** your code to determine most likely places needing parallelization or optimization.

- **Parallelize** your code by starting with the most time consuming parts and check for correctness.

- **Optimize** your code to improve observed speed-up from parallelization.
PROFILING SEQUENTIAL CODE

Profile Your Code
Obtain detailed information about how the code ran.

- Total runtime
- Runtime of individual routines
- Hardware counters

Identify the portions of code that took the longest to run. We want to focus on these “hotspots” when parallelizing.

Lab Code: Laplace Heat Transfer
Total Runtime: 39.43 seconds

- swap: 19.04s
- calcNext: 21.49s
PROFILING SEQUENTIAL CODE

CPU Details

- We can see that there are two places that our code is spending most of its time
- 21.49 seconds in the “calcNext” function
- 19.04 seconds in a memcpy function
- The c_mcopy8 that we see is actually a compiler optimization that is being applied to our “swap” function
We are also able to select the different elements in the CPU Details by double-clicking to open the associated source code.

Here we have selected the “calcNext:37” element, which opened up our code to show the exact line (line 37) that is associated with that element.
OPENACC PARALLEL DIRECTION
OPENACC PARALLEL DIRECTIVE
Expressing parallelism

```c
#pragma acc parallel
{
    When encountering the `parallel` directive, the compiler will generate
    1 or more parallel `gangs`, which execute redundantly.
}
```

![Diagram showing parallel gangs executing redundantly]
OPENACC PARALLEL DIRECTIVE

Expressing parallelism

```
#pragma acc parallel
{
for(int i = 0; i < N; i++)
{
  // Do Something
}
}
```

This loop will be executed redundantly on each gang
OPENACC PARALLEL DIRECTIVE
Expressing parallelism

#pragma acc parallel
{
  for(int i = 0; i < N; i++)
  {
    // Do Something
  }
}

This means that each gang will execute the entire loop.
OPENACC PARALLEL DIRECTIVE
Parallelizing a single loop

C/C++

```c
#pragma acc parallel
{
    #pragma acc loop
    for(int i = 0; j < N; i++)
        a[i] = 0;
}
```

- Use a `parallel` directive to mark a region of code where you want parallel execution to occur.
- This parallel region is marked by curly braces in C/C++ or a start and end directive in Fortran.
- The `loop` directive is used to instruct the compiler to parallelize the iterations of the next loop to run across the parallel gangs.

Fortran

```fortran
!$acc parallel
!$acc loop
    do i = 1, N
        a(i) = 0
    end do
!$acc end parallel
```

- Use a `loop` directive to parallelize the iterations of the next loop.
OPENACC PARALLEL DIRECTIVE
Parallelizing a single loop

- This pattern is so common that you can do all of this in a single line of code

- In this example, the parallel loop directive applies to the next loop

- This directive both marks the region for parallel execution and distributes the iterations of the loop.

- When applied to a loop with a data dependency, parallel loop may produce incorrect results

C/C++

```c
#pragma acc parallel loop
for(int i = 0; j < N; i++)
a[i] = 0;
```

Fortran

```fortran
!$acc parallel loop
do i = 1, N
   a(i) = 0
end do
```
OPENACC PARALLEL DIRECTIVE

Expressing parallelism

```c
#pragma acc parallel
{
  #pragma acc loop
  for(int i = 0; i < N; i++)
  {
    // Do Something
  }
}
```

The *loop* directive informs the compiler which loops to parallelize.
OPENACC PARALLEL LOOP DIRECTIVE
Parallelizing many loops

- To parallelize multiple loops, each loop should be accompanied by a parallel directive.

- Each parallel loop can have different loop boundaries and loop optimizations.

- Each parallel loop can be parallelized in a different way.

- This is the recommended way to parallelize multiple loops. Attempting to parallelize multiple loops within the same parallel region may give performance issues or unexpected results.

```c
#pragma acc parallel loop
for(int i = 0; i < N; i++)
    a[i] = 0;

#pragma acc parallel loop
for(int j = 0; j < M; j++)
    b[j] = 0;
```
PARALLELIZE WITH OPENACC PARALLEL LOOP

```c
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++;
}
```

Parallelize first loop nest, max reduction required.

Parallelize second loop.

We didn’t detail how to parallelize the loops, just which loops to parallelize.
BUILDING THE CODE (GPU)

$ pgcc -f -acc -ta=tesla:managed,cc70 -Minfo=accel laplace2d_uvm.c
main:

63, Accelerator kernel generated
Generating Tesla code
64, #pragma acc loop gang /* blockIdx.x */
    Generating reduction(max:error)
66, #pragma acc loop vector(128) /* threadIdx.x */
63, Generating implicit copyin(A[:])
Generating implicit copyout(Anew[:])
Generating implicit copy(error)
66, Loop is parallelizable
74, Accelerator kernel generated
Generating Tesla code
75, #pragma acc loop gang /* blockIdx.x */
77, #pragma acc loop vector(128) /* threadIdx.x */
74, Generating implicit copyin(Anew[:])
Generating implicit copyout(A[:])
77, Loop is parallelizable
BUILDING THE CODE (MULTICORE)

$ pgcc -fast -acc -ta=multicore -Minfo=accel laplace2d_uvm.c
main:
  63, Generating Multicore code
  64, #pragma acc loop gang
  64, Accelerator restriction: size of the GPU copy of Anew,A is unknown
  Generating reduction(max:error)
  66, Loop is parallelizable
  74, Generating Multicore code
  75, #pragma acc loop gang
  75, Accelerator restriction: size of the GPU copy of Anew,A is unknown
  77, Loop is parallelizable
OPENACC SPEED-UP

- SERIAL: 1.00X
- MULTICORE: 3.23X
- V100: 41.80X
BUILDING THE CODE (GPU)

$ pgcc -fast -acc -ta=tesla,cc70 -Minfo=accel laplace2d_uvm.c
PGC-S-0155-Compiler failed to translate accelerator region (see -Minfo messages):
Could not find allocated-variable index for symbol (laplace2d_uvm.c: 63)
PGC-S-0155-Compiler failed to translate accelerator region (see -Minfo messages):
Could not find allocated-variable index for symbol (laplace2d_uvm.c: 74)

main:
   63, Accelerator kernel generated
       Generating Tesla code
       63, Generating reduction(max:error)
       64, #pragma acc loop gang /* blockIdx.x */
       66, #pragma acc loop vector(128) /* threadIdx.x */
   64, Accelerator restriction: size of the GPU copy of Anew,A is unknown
   66, Loop is parallelizable
   74, Accelerator kernel generated
       Generating Tesla code
       75, #pragma acc loop gang /* blockIdx.x */
       77, #pragma acc loop vector(128) /* threadIdx.x */
   75, Accelerator restriction: size of the GPU copy of Anew,A is unknown
   77, Loop is parallelizable
OPTIMIZE DATA MOVEMENT
EXPLICIT MEMORY MANAGEMENT

Key problems

- Many parallel accelerators (such as devices) have a separate memory pool from the host
- These separate memories can become out-of-sync and contain completely different data
- Transferring between these two memories can be a very time consuming process
OPENACC DATA DIRECTIVE

Definition

- The data directive defines a lifetime for data on the device
- During the region data should be thought of as residing on the accelerator
- Data clauses allow the programmer to control the allocation and movement of data

```c
#pragma acc data clauses
{
  < Sequential and/or Parallel code >
}

!$acc data clauses
< Sequential and/or Parallel code >
!$acc end data
```
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.

**Principal use:** For many important data structures in your code, this is a logical default to input, modify and return the data.

**copyin(list)**
Allocates memory on GPU and copies data from host to GPU when entering region.

**Principal use:** Think of this like an array that you would use as just an input to a subroutine.

**copyout(list)**
Allocates memory on GPU and copies data to the host when exiting region.

**Principal use:** A result that isn’t overwriting the input data structure.

**create(list)**
Allocates memory on GPU but does not copy.

**Principal use:** Temporary arrays.
ARRAY SHAPING

- Sometimes the compiler needs help understanding the *shape* of an array
- The first number is the start index of the array
- In C/C++, the second number is how much data is to be transferred
- In Fortran, the second number is the ending index

```c/c++
copy(array[starting_index:length])
```

```fortran
copy(array(starting_index:ending_index))
```
ARRAY SHAPING (CONT.)
Multi-dimensional Array shaping

Both of these examples copy a 2D array to the device

C/C++
copy(array[0:N][0:M])

Fortran
copy(array(1:N, 1:M))
ARRAY SHAPING (CONT.)

Partial Arrays

```
copy(array[i*N/4:N/4])
```

C/C++

Both of these examples copy only ¼ of the full array

```
copy(array(i*N/4:i*N/4+N/4))
```

Fortran
STRUCTURED DATA DIRECTIVE

Example

```c
#pragma acc data copyin(a[0:N],b[0:N]) copyout(c[0:N])
{
    #pragma acc parallel loop
    for(int i = 0; i < N; i++){
        c[i] = a[i] + b[i];
    }
}
```

Action

<table>
<thead>
<tr>
<th>Host Memory</th>
<th>Device memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>C'</td>
</tr>
<tr>
<td>C'</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>A</td>
<td>C'</td>
</tr>
</tbody>
</table>

Allocate A on device
Copy A from CPU to device
Allocate B on device
Copy B from CPU to device
Allocate C on device
Execute loop on device
Copy C from device to CPU
Deallocate C from device
Deallocate B from device
Deallocate A from device
OPTIMIZED DATA MOVEMENT

```c
#pragma acc data copy(A[:n*m]) copyin(Anew[:n*m])
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]));
        }
    }

#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.
Copy initial condition of Anew, but not final value.
REBUILD THE CODE

pgcc -fast -ta=tesla -Minfo=accel laplace2d_uvm.c

main:

60, Generating copy(A[:m*n])
   Generating copyin(Anew[:m*n])

64, Accelerator kernel generated
   Generating Tesla code
   64, Generating reduction(max:error)
   65, #pragma acc loop gang /* blockIdx.x */
   67, #pragma acc loop vector(128) /* threadIdx.x */

67, Loop is parallelizable

75, Accelerator kernel generated
   Generating Tesla code
   76, #pragma acc loop gang /* blockIdx.x */
   78, #pragma acc loop vector(128) /* threadIdx.x */

78, Loop is parallelizable

Now data movement only happens at our data region.
OPENACC SPEED-UP

Speed-up

- SERIAL: 1.00X
- MULTICORE: 3.23X
- V100: 41.80X
- V100 (DATA): 42.99X
DATA SYNCHRONIZATION
OPENACC UPDATE DIRECTIVE

**update**: Explicitly transfers data between the host and the device

Useful when you want to synchronize data in the middle of a data region

Clauses:

**self**: makes host data agree with device data

**device**: makes device data agree with host data

```c
#pragma acc update self(x[0:count])
#pragma acc update device(x[0:count])
```

C/C++

```fortran
!$acc update self(x(1:end_index))
!$acc update device(x(1:end_index))
```

Fortran
OPENACC UPDATE DIRECTIVE

The data must exist on both the CPU and device for the update directive to work.

```
#pragma acc update device(A[0:N])
```

```
#pragma acc update self(A[0:N])
```
SYNCHRONIZE DATA WITH UPDATE

Inside the `initialize` function we alter the host copy of ‘A’

This means that after calling `initialize` the host and device copy of ‘A’ are out-of-sync

We use the `update` directive with the `device` clause to update the device copy of ‘A’

Without the `update` directive later compute regions will use incorrect data.
FURTHER OPTIMIZATIONS
PROFILING GPU CODE (PGPROF)

Using PGPROF to profile GPU code

- PGPROF presents far more information when running on a GPU
- We can view CPU Details, GPU Details, a Timeline, and even do Analysis of the performance
PROFILING GPU CODE (PGPROF)

Using PGPROF to profile GPU code

- **Memcpy(HtoD):** This includes data transfers from the Host to the Device (CPU to GPU)
- **Memcpy(DtoH):** These are data transfers from the Device to the Host (GPU to CPU)
- **Compute:** These are our computational functions. We can see our calcNext and swap function
LOOP OPTIMIZATIONS
COLLAPSE CLAUSE

- `collapse( N )`
- Combine the next N tightly nested loops
- Can turn a multidimensional loop nest into a single-dimension loop
- This can be extremely useful for increasing memory locality, as well as creating larger loops to expose more parallelism

```c
#pragma acc parallel loop collapse(2)
for( i = 0; i < size; i++ )
  for( j = 0; j < size; j++ )
    double tmp = 0.0f;
    #pragma acc loop reduction(+:tmp)
    for( k = 0; k < size; k++ )
      tmp += a[i][k] * b[k][j];
    c[i][j] = tmp;
```
#pragma acc parallel loop collapse(2)

```c
for ( i = 0; i < 4; i++ )
    for ( j = 0; j < 4; j++ )
        array[i][j] = 0.0f;
```
TILE CLAUSE

- **tile (x, y, z, ...)**
- Breaks multidimensional loops into “tiles” or “blocks”
- Can increase data locality in some codes
- Will be able to execute multiple “tiles” simultaneously

```c
#pragma acc kernels loop tile(32, 32)
for( i = 0; i < size; i++ )
  for( j = 0; j < size; j++ )
    for( k = 0; k < size; k++ )
      c[i][j] += a[i][k] * b[k][j];
```
#pragma acc kernels loop tile(2,2)
for(int x = 0; x < 4; x++){
    for(int y = 0; y < 4; y++){
        array[x][y]++;
    }
}
GANG WORKER VECTOR

- Gang / Worker / Vector defines the various levels of parallelism we can achieve with OpenACC.

- This parallelism is most useful when parallelizing multi-dimensional loop nests.

- OpenACC allows us to define a generic Gang / Worker / Vector model that will be applicable to a variety of hardware, but we will focus a little bit on a GPU specific implementation.
#pragma acc data copy(A[:n*m]) copyin(Anew[:n*m])
while (err > tol && iter < iter_max) {
    err = 0.0;

    #pragma acc parallel loop reduction(max:err) tile(32,32)
    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 tile(32,32)
    for (int j = 1; j < n-1; j++) {
        for (int i = 1; i < m-1; i++) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}

Create 32x32 tiles of the loops to better exploit data locality.
OPENACC SPEED-UP

<table>
<thead>
<tr>
<th>Speed-Up</th>
<th>SERIAL</th>
<th>MULTICORE</th>
<th>V100</th>
<th>V100 (DATA)</th>
<th>V100 (TILE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed-up</td>
<td>1.00X</td>
<td>3.23X</td>
<td>41.8X</td>
<td>42.99X</td>
<td>54.25X</td>
</tr>
</tbody>
</table>
GPU LOOP OPTIMIZATION:
RULES OF THUMB

▪ It is rarely a good idea to set the number of gangs in your code, let the compiler decide.

▪ Most of the time you can effectively tune a loop nest by adjusting only the vector length.

▪ It is rare to use a worker loop. When the vector length is very short, a worker loop can increase the parallelism in your gang.

▪ When possible, the vector loop should step through your arrays

▪ Use the device_type clause to ensure that tuning for one architecture doesn’t negatively affect other architectures.
NVIDIA RESOURCES

NVIDIA Developer  
https://developer.nvidia.com/

CUDA Toolkit  

GPU Accelerated Libraries  

OpenACC Resources  
https://www.openacc.org/resources

PGI Community Edition Compiler  
https://www.pgroup.com/products/community.htm