Cloud Innovator
MEGAZONE

Megazone is delivering values through the business expertise and experiences built from the Platform Service, Digital Marketing and Digital Service areas

OpenACC
More Science, Less Programming

OpenACC

More Science Less Programming

Matilda Lab / Hybrid AI Center / Megazone Cloud Corp.
Sungho Kim, Ph.D.

2023. 12.

https://developer.nvidia.com/intro-to-openacc-course-2016

Contents

Cloud Innovator
MEGAZONE

Megazone is delivering values through the business expertise and experiences built from the Platform Service, Digital Marketing and Digital Service areas

OpenACC
More Science, Less Programming

GPU Programming for HPC

Cloud Innovator MEGAZONE

OpenACC
More Science, Less Programming

N-Ways to GPU Programming:
Building Scalable GPU-Accelerated Applications.

- Math Libraries | Standard Languages | Directives | CUDA

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

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

GPU Accelerated C++ and Fortran

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

...

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

...

}
```

Incremental Performance Optimization with Directives

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

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

Maximize GPU Performance with CUDA C++/Fortran

GPU Accelerated Math Libraries

GPU Programming for HPC

- NVIDIA GPUs can be programmed much like CPUs.

1. Start by substituting GPU-optimized math libraries.
2. Add additional acceleration using the standard C++ parallel algorithms and Fortran language features.
3. Use pragmas and directives to fill any standard language gaps, and finally, optimize performance with CUDA®.

GPU Programming for HPC

- Libraries
  - Drop-in GPU-accelerated libraries are an easy replacement for CPU libraries.
  - Multi-GPU and multi-node aware, NVIDIA GPU-accelerated libraries provide the best performance for the most common patterns in HPC applications. Select from a wide variety of libraries optimized for commonly used computing operations.
- Standard Languages
  - Parallel features in standard C++ and Fortran can map routines to either the cores of a multi-core CPU or a GPU.
  - The NVIDIA C++17 compilers add support for execution policies on the standard template library (STL), and the NVIDIA Fortran 2008 compiler's DO CONCURRENT construct allows loops to iterate without interdependencies.

GPU Programming for HPC

- Directives
  - Directive-based programming models provide an easy on-ramp to parallel computing on GPUs, CPUs, and other devices.
  - If standard languages don't have the flexibility or features you need to get good performance, augment with directives and remain portable to other compilers and platforms.
- CUDA
  - CUDA is a parallel computing platform and programming model designed to deliver the most flexibility and performance for GPU-accelerated applications.
  - To maximize performance and flexibility, get the most out of the GPU hardware by coding directly in CUDA C/C++ or CUDA Fortran.

Math Libraries

Math Libraries

cuBLAS

BF16, TF32 and FP64 Tensor Cores

cuSPARSE

Sparse MMA Tensor Core, Increased memory BW, Shared Memory and L2

cuTENSOR

BF16, TF32 and FP64 Tensor Cores

cuSOLVER

BF16, TF32 and FP64 Tensor Cores

cuFFT

Increased memory BW, Shared Memory and L2

CUDA Math API

BF16 Support

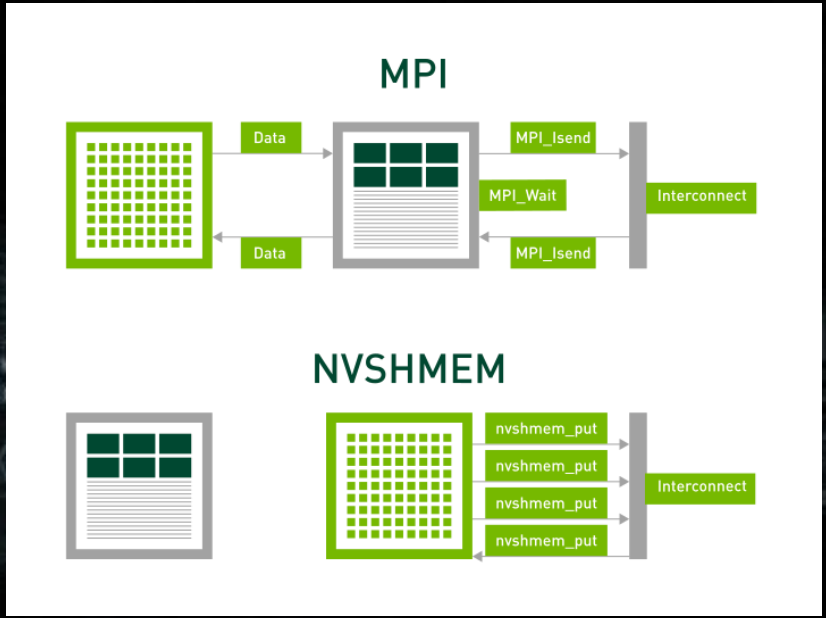
Communication Libraries

Communication Libraries

- Performance-optimized multi-GPU and multi-node communication primitives.
- **NVSHMEM**
  - NVSHMEM™ is a parallel programming interface based on OpenSHMEM that provides efficient and scalable communication for NVIDIA GPU clusters. NVSHMEM creates a global address space for data that spans the memory of multiple GPUs and can be accessed with fine-grained GPU-initiated operations, CPU-initiated operations, and operations on CUDA® streams.
- **NCCL**
  - Open-source library for fast multi-GPU, multi-node communications that maximizes bandwidth while maintaining low latency.

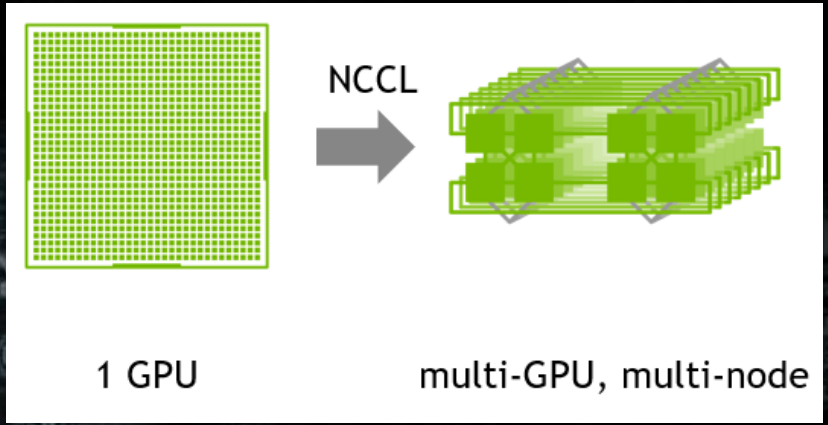
NVIDIA NVSHMEM

- NVSHMEM™ is a parallel programming interface based on OpenSHMEM that provides efficient and scalable communication for NVIDIA GPU clusters.
- NVSHMEM creates a global address space for data that spans the memory of multiple GPUs and can be accessed with fine-grained GPU-initiated operations, CPU-initiated operations, and operations on CUDA® streams.



NVIDIA NCCL

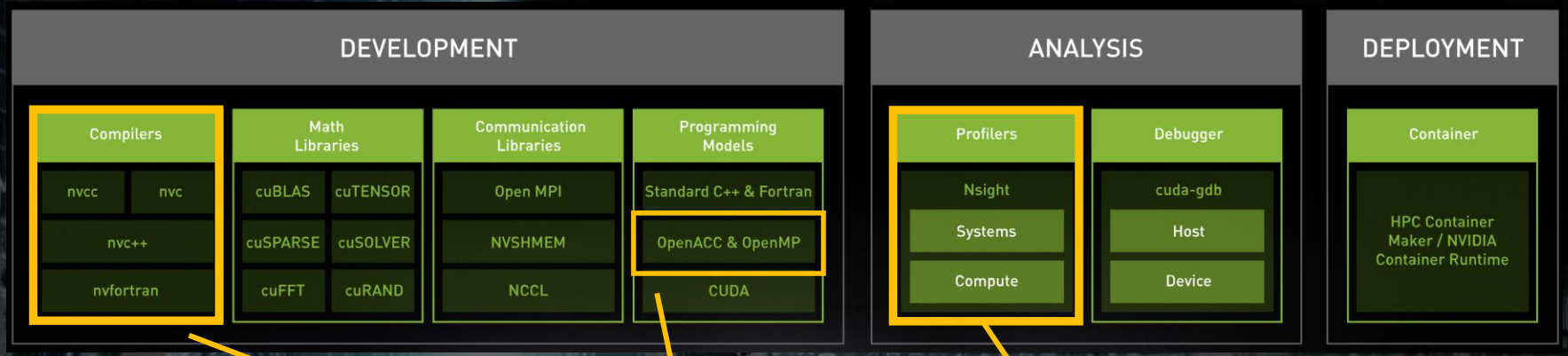
- The NVIDIA Collective Communication Library (NCCL) implements multi-GPU and multi-node communication primitives optimized for NVIDIA GPUs and Networking.
- NCCL provides routines such as all-gather, all-reduce, broadcast, reduce, reduce-scatter as well as point-to-point send and receive that are optimized to achieve high bandwidth and low latency over PCIe and NVLink high-speed interconnects within a node and over NVIDIA Mellanox Network across nodes.



NVIDIA NCCL

- **Performance**
  - NCCL conveniently removes the need for developers to optimize their applications for specific machines.
  - NCCL provides fast collectives over multiple GPUs both within and across nodes.
- **Ease of Programming**
  - NCCL uses a simple C API, which can be easily accessed from a variety of programming languages.
  - NCCL closely follows the popular collectives API defined by MPI (Message Passing Interface).
- **Compatibility**
  - NCCL is compatible with virtually any multi-GPU parallelization model, such as: single-threaded, multi-threaded (using one thread per GPU) and multi-process (MPI combined with multi-threaded operation on GPUs).
- **Key Features**
  - Automatic topology detection for high bandwidth paths on AMD, ARM, PCI Gen4 and IB HDR
  - Up to 2x peak bandwidth with in-network all reduce operations utilizing SHARPV2
  - Graph search for the optimal set of rings and trees with the highest bandwidth and lowest latency
  - Support multi-threaded and multi-process applications
  - InfiniBand verbs, libfabric, RoCE and IP Socket internode communication
  - Reroute traffic and alleviate congested ports with InfiniBand Adaptive routing
- Leading deep learning frameworks such as Caffe2, Chainer, MxNet, PyTorch and TensorFlow have integrated NCCL to accelerate deep learning training on multi-GPU multi-node systems.

Back to HPC SDK



Have time to learn this tool.

This course covers some of these parts.

Cloud Innovator
MEGAZONE

Megazone is delivering values through the business expertise and experiences built from the Platform Service, Digital Marketing and Digital Service areas

OpenACC
More Science, Less Programming

OpenACC Basic Training Course

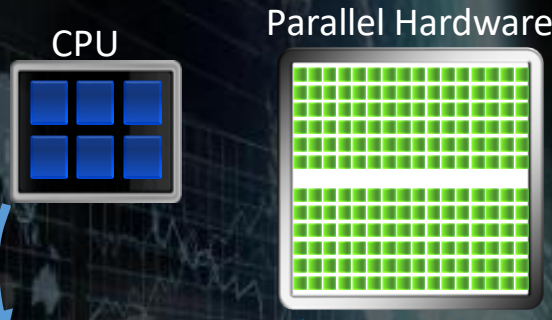
OpenACC is

a directives-based **parallel programming Model** designed for **performance** and **portability** on CPUs and accelerators for HPC.

Add Simple Compiler Directive

```
main()
 {
  <serial code>
  #pragma acc kernels
  {
    <parallel code>
  }
}
```

OpenACC Directives




```
int main(){

 <sequential code>

 #pragma acc kernels
 {
  <parallel code>
 }

}
```

Compiler Directive

1. Simple compiler hints from programmer
2. Compiler generates parallel threaded code
3. Ignorant compiler just sees some comments.

Incremental

Single Source

Low Learning Curve

More on this later!

Cloud Innovator MEGAZONE

OpenACC
More Science, Less Programming

Single code for multiple platforms

OpenACC - Performance Portable Programming Model for HPC

OpenPOWER

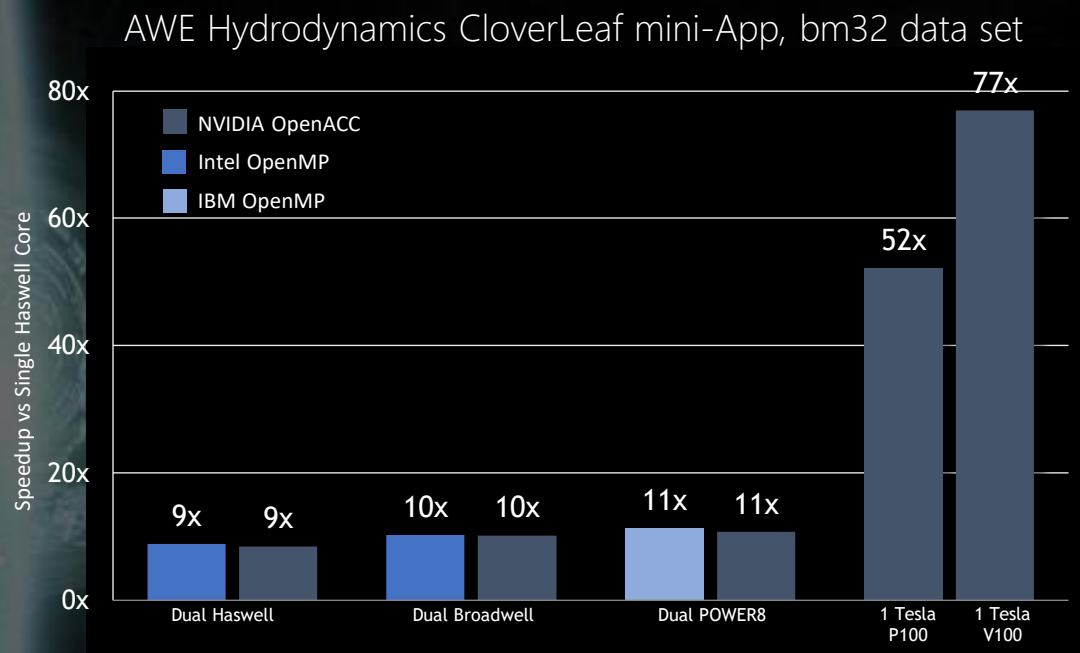
Sunway

x86 CPU

x86 Xeon Phi

NVIDIA GPU

PEZY-SC



Systems: Haswell: 2x16 core Haswell server, four K80s, CentOS 7.2 (perf-hsw10), Broadwell: 2x20 core Broadwell server, eight P100s (dgx1-prd-01), Minsky: POWER8+NVLINK, four P100s, RHEL 7.3 (gsn1).
Compilers: Intel 17.0, IBM XL 13.1.3, PGI 16.10, KNL: Compiler version: 17.0.1 20161005,
Benchmark: CloverLeaf v1.3 downloaded from http://uk-mac.github.io/CloverLeaf the week of November 7 2016; CloverlLeaf_Serial; CloverLeaf_ref (MPI+OpenMP); CloverLeaf_OpenACC (MPI+OpenACC)
Data compiled by PGI November 2016, Volta data collected June 2017

Cloud Innovator MEGAZONE

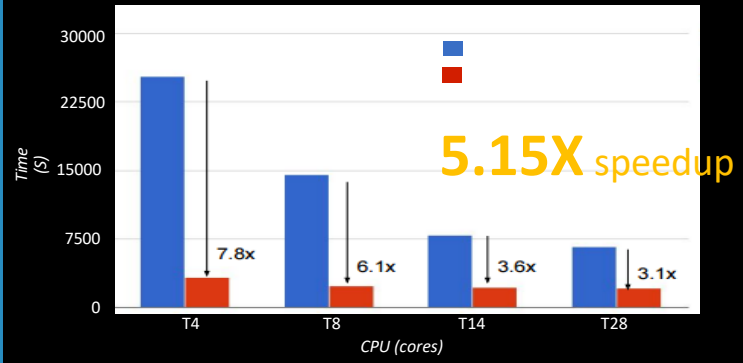
OpenACC
More Science, Less Programming

Top HPC apps adopting OpenACC

ANSYS Fluent • Gaussian • VASP • GTC • XGC • ACME • FLASH • LSDalton • COSMO • ELEPHANT • RAMSES • ICON • ORB5

ANSYS Fluent

ANSYS Fluent R18.0 Radiation Solver

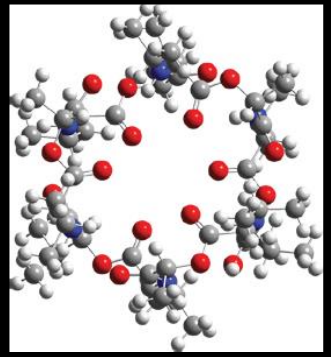


5.15X speedup

CPU: (Haswell EP) Intel(R) Xeon(R) CPU E5-2695 v3 @2.30GHz, 2 sockets, 28 cores
GPU: Tesla K80 12+12 GB, Driver 346.46

Gaussian 16

Valinomycin wB97xD/6-311+(2d,p) Freq

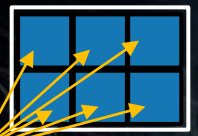


2.25X speedup

Hardware: HPE server with dual Intel Xeon E5-2698 v3 CPUs (2.30GHz ; 16 cores/chip), 256GB memory and 4 Tesla K80 dual GPU boards (boost clocks: MEM 2505 and SM 875). Gaussian source code compiled with PGI Accelerator Compilers (16.5) with OpenACC (2.5 standard).

Familiar to OpenMP Programmers

CPU

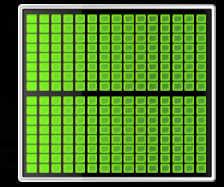


```
main() {
  double pi = 0.0; long i;

  #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);
}
```

OpenMP

CPU

Parallel Hardware



```
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

3 Ways to Accelerate Applications

Applications

Libraries	Compiler Directives	Programming Languages
Easy to use Most Performance	Easy to use Portable code OpenACC	Most Performance Most Flexibility **CUDA, OpenCL**

OPENACC: Key Advantages

- **High-level**.
  - Minimal modifications to the code. Less than with OpenCL, CUDA, etc. Non-GPU programmers can play along.
- **Single source**.
  - No GPU-specific code. Compile the same program for accelerators or serial.
- **Efficient**.
  - Experience shows very favorable comparison to low-level implementations of same algorithms.
- **Performance portable**.
  - Supports CPUs, GPU accelerators and co-processors from multiple vendors, current and future versions.
- **Incremental**.
  - Developers can port and tune parts of their application as resources and profiling dictates. No wholesale rewrite required. Which can be *quick.*

True Open Standard

- Full OpenACC 3.3 spec. available at https://www.openacc.org/specification
- Quick reference card and guides: https://www.openacc.org/resources
- Tools of NVIDIA, AMD, GCC, etc.: https://www.openacc.org/tools
- GCC 13,12,… supports OpenACC2.6 : https://gcc.gnu.org/wiki/OpenACC
- Best free option is NVIDIA HPC SDK: https://developer.nvidia.com/hpc-sdk

OpenACC.org Members

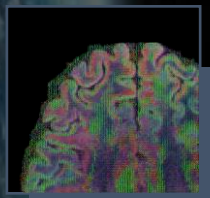
Hewlett Packard Enterprise
allinea
AMD
NVIDIA
SIEMENS Ingenuity for life
INDIANA UNIVERSITY
国家超级计算无锡中心 National Supercomputing Center In Wuxi
RIKEN
筑波大学 University of Tsukuba
Score-P
NOAA
HZDR
BROOKHAVEN NATIONAL LABORATORY
東京工業大学 Tokyo Institute of Technology
KAUST
HELMHOLTZ ZENTRUM DRESDEN ROSSENDORF
UNIVERSITY OF DELAWARE
epcc
TOTAL
OAK RIDGE National Laboratory
VirginiaTech
Stony Brook University

Cloud Innovator
MEGAZONE

OpenACC
More Science, Less Programming

LSDalton

Quantum Chemistry
Aarhus University
12X speedup
1 week

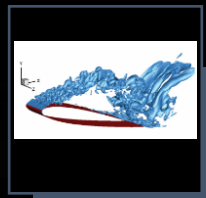


PowerGrid

Medical Imaging
University of Illinois
40 days to
2 hours

COSMO

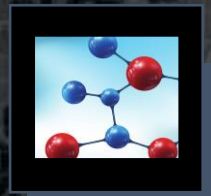
Weather and Climate
MeteoSwiss, CSCS
4X speedup
3X energy efficiency



INCOMP3D

CFD
NC State University

4X speedup

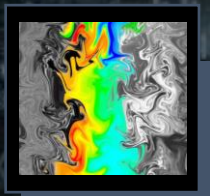


NekCEM

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

Example Code

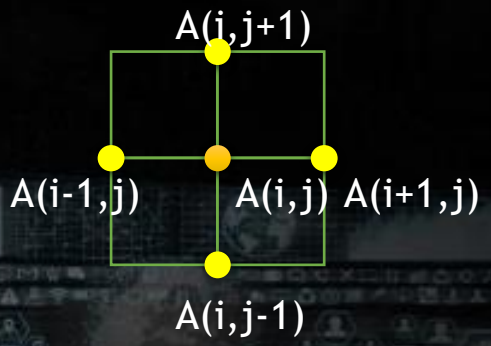
Foundation Exercise: Laplace Solver

Introduction to lab code - visual

- I've been using this for MPI, OpenMP and now OpenACC. It is a great simulation problem, not rigged for OpenACC.
  - In this most basic form, it solves the Laplace equation: $\nabla^2 f(x, y) = 0$
- The Laplace Equation applies to many physical problems, including: electrostatics, fluid flow, and temperature
- For temperature, it is the Steady State Heat Equation:

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: $\nabla^2 f(x,y) = 0$



$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

Serial Code Implementation

C

```
for(i = 1; i <= ROWS; i++) {
     for(j = 1; j <= COLUMNS; j++) {
          Anew[i][j] = 0.25 * (A[i+1][j] + A[i-1][j] + A[i][j+1] + A[i][j-1]);
     }
}
```

Fortran

```
do j=1,columns
     do i=1,rows
          Anew(i,j)= 0.25 * (A(i+1,j)+A(i-1,j) + A(i,j+1)+A(i,j-1) )
     enddo
enddo
```

Serial C Code (kernel)

```
while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

    for(i = 1; i <= ROWS; i++) {
        for(j = 1; j <= COLUMNS; j++) {
            Anew[i][j] = 0.25 * (A[i+1][j] + A[i-1][j] + A[i][j+1] + A[i][j-1]);
        }
    }

    dt = 0.0;

    for(i = 1; i <= ROWS; i++){
        for(j = 1; j <= COLUMNS; j++){
        dt = fmax( fabs(Anew[i][j]-A[i][j]), dt);
        A[i][j] = Anew[i][j];
        }
    }

    if((iteration % 100) == 0) {
        track_progress(iteration);
    }

    iteration++;

}
```

Done?

Calculate

Update temp array and find max change

Output

Serial C Code Subroutines

```
void initialize(){

    int i,j;

    for(i = 0; i <= ROWS+1; i++){
        for (j = 0; j <= COLUMNS+1; j++){
            A[i][j] = 0.0;
        }
    }

    // these boundary conditions never change throughout run

    // set left side to 0 and right to a linear increase
    for(i = 0; i <= ROWS+1; i++) {
        A[i][0] = 0.0;
        A[i][COLUMNS+1] = (100.0/ROWS)*i;
    }

    // set top to 0 and bottom to linear increase
    for(j = 0; j <= COLUMNS+1; j++) {
        A[0][j] = 0.0;
        A[ROWS+1][j] = (100.0/COLUMNS)*j;
    }
}
```

```
void track_progress(int iteration) {

  int i;

  printf("-- Iteration: %d --\n", iteration);
  for(i = ROWS-5; i <= ROWS; i++) {
     printf("[%d,%d]: %5.2f ", i, i,Anew[i][i]);
  }
  printf("\n");
}
```

BCs could run from 0 to ROWS+1 or from 1 to ROWS. We chose the former.

Cloud Innovator MEGAZONE

Whole C Code

OpenACC
More Science, Less Programming

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <sys/time.h>

// size of plate
#define COLUMNS    1000
#define ROWS       1000

// largest permitted change in temp (This value takes about 3400 steps)
#define MAX_TEMP_ERROR 0.01

double Temperature[ROWS+2][COLUMNS+2];      // temperature grid
double Temperature_last[ROWS+2][COLUMNS+2]; // temperature grid from last iteration

//   helper routines
void initialize();
void track_progress(int iter);

int main(int argc, char *argv[]) {

    int i, j;                                           // grid indexes
    int max_iterations;                                 // number of iterations
    int iteration=1;                                    // current iteration
    double dt=100;                                      // largest change in t
    struct timeval start_time, stop_time, elapsed_time;  // timers

    printf("Maximum iterations [100-4000]?\n");
    scanf("%d", &max_iterations);

    gettimeofday(&start_time,NULL); // Unix timer

    initialize();                   // initialize Temp_last including boundary conditions

    // do until error is minimal or until max steps
    while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

        // main calculation: average my four neighbors
        for(i = 1; i <= ROWS; i++) {
            for(j = 1; j <= COLUMNS; j++) {
                    Anew[i][j] = 0.25 * (A[i+1][j] + A[i-1][j] + A[i][j+1] + A[i][j-1]);
            }
        }

        dt = 0.0; // reset largest temperature change

        // copy grid to old grid for next iteration and find latest dt
        for(i = 1; i <= ROWS; i++){
            for(j = 1; j <= COLUMNS; j++){
              dt = fmax( fabs(Anew[i][j]-A[i][j]), dt);
              A[i][j] = Anew[i][j];
            }
        }

        // periodically print test values
        if((iteration % 100) == 0) {
            track_progress(iteration);
        }

        iteration++;
    }

    gettimeofday(&stop_time,NULL);
    timersub(&stop_time, &start_time, &elapsed_time); // Unix time subtract routine

    printf("\nMax error at iteration %d was %f\n", iteration-1, dt);
    printf("Total time was %f seconds.\n", elapsed_time.tv_sec+elapsed_time.tv_usec/1000000.0);

}

// initialize plate and boundary conditions
// Temp_last is used to to start first iteration
void initialize(){

    int i,j;

    for(i = 0; i <= ROWS+1; i++){
        for (j = 0; j <= COLUMNS+1; j++){
            A[i][j] = 0.0;
        }
    }

    // these boundary conditions never change throughout run

    // set left side to 0 and right to a linear increase
    for(i = 0; i <= ROWS+1; i++) {
        A[i][0] = 0.0;
        A[i][COLUMNS+1] = (100.0/ROWS)*i;
    }

    // set top to 0 and bottom to linear increase
    for(j = 0; j <= COLUMNS+1; j++) {
        A[0][j] = 0.0;
        A[ROWS+1][j] = (100.0/COLUMNS)*j;
    }
}

// print diagonal in bottom right corner where most action is
void track_progress(int iteration) {

    int i;

    printf("---------- Iteration number: %d ------------\n", iteration);
    for(i = ROWS-5; i <= ROWS; i++) {
        printf("[%d,%d]: %5.2f  ", i, i, Anew[i][i]);
    }
    printf("\n");
}
```

JACOBI ITERATION: Serial C Code

```
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++;
}
```

Iterate Until converged

Iterate across matrix elements

Calculate new value from neighbors

Compute max error for convergence

Swap Input/output arrays

Serial Fortran Code

```
do while ( dt > max_temp_error .and. iteration <= max_iterations)

   do j=1,columns
      do i=1,rows
         Anew(i,j)=0.25*( A(i+1,j)+A(i-1,j)+A(i,j+1)+A(i,j-1) )
      enddo
   enddo

   dt=0.0

   do j=1,columns
      do i=1,rows
         dt = max( abs(Anew(i,j) - A(i,j)), dt )
         A(i,j) = Anew(i,j)
      enddo
   enddo

   if( mod(iteration,100).eq.0 ) then
      call track_progress(Anew, iteration)
   endif

   iteration = iteration+1

enddo
```

Done?

Calculate

Update temp array and find max change

Output

Serial Fortran Code Subroutines

```
subroutine initialize( A )
      implicit none

      integer, parameter              :: columns=1000
      integer, parameter              :: rows=1000
      integer                         :: i,j

      double precision, dimension(0:rows+1,0:columns+1) ::
temperature_last

      A = 0.0

      !these boundary conditions never change throughout run

      !set left side to 0 and right to linear increase
      do i=0,rows+1
         A(i,0) = 0.0
         A(i,columns+1) = (100.0/rows) * i
      enddo

      !set top to 0 and bottom to linear increase
      do j=0,columns+1
         A(0,j) = 0.0
         A(rows+1,j) = ((100.0)/columns) * j
      enddo

end subroutine initialize
```

```
subroutine track_progress(Anew, iteration)
      implicit none

      integer, parameter              :: columns=1000
      integer, parameter              :: rows=1000
      integer                         :: i,iteration

      double precision, dimension(0:rows+1,0:columns+1) :: Anew

      print *, '---------- Iteration number: ', iteration, ' ---------------'
      do i=5,0,-1
         write (*,'("("i4,",",i4,"):",f6.2,"  ")',advance='no'), &
                  rows-i,columns-i,Anew(rows-i,columns-i)
      enddo
      print *
```

Cloud Innovator MEGAZONE

OpenACC
More Science, Less Programming

Whole Fortran Code

```
program serial
      implicit none

      !Size of plate
      integer, parameter              :: columns=1000
      integer, parameter              :: rows=1000
      double precision, parameter     :: max_temp_error=0.01

      integer                         :: i, j, max_iterations, iteration=1
      double precision                :: dt=100.0
      real                            :: start_time, stop_time

      double precision, dimension(0:rows+1,0:columns+1) :: Anew, A

      print*, 'Maximum iterations [100-4000]?'
      read*,   max_iterations

      call cpu_time(start_time)       !Fortran timer

      call initialize( A )

      !do until error is minimal or until maximum steps
      do while ( dt > max_temp_error .and. iteration <= max_iterations)

         do j=1,columns
             do i=1,rows
                Anew(i,j)=0.25*(A(i+1,j)+A(i-1,j)+A(i,j+1)+A(i,j-1) )
             enddo
         enddo

         dt=0.0

         !copy grid to old grid for next iteration and find max change
         do j=1,columns
             do i=1,rows
                dt = max( abs(Anew(i,j) - A(i,j)), dt )
                A(i,j) = Anew(i,j)
             enddo
         enddo

         !periodically print test values
         if( mod(iteration,100).eq.0 ) then
             call track_progress( Anew, iteration)
         endif

         iteration = iteration+1

      enddo

      call cpu_time(stop_time)

      print*, 'Max error at iteration ', iteration-1, ' was ',dt
      print*, 'Total time was ',stop_time-start_time, ' seconds.'

end program serial
```

```
! initialize plate and boundery conditions
! temp_last is used to to start first iteration
subroutine initialize( A )
      implicit none

      integer, parameter              :: columns=1000
      integer, parameter              :: rows=1000
      integer                         :: i,j

      double precision, dimension(0:rows+1,0:columns+1) :: A

      A = 0.0

      !these boundary conditions never change throughout run

      !set left side to 0 and right to linear increase
      do i=0,rows+1
         A(i,0) = 0.0
         A(i,columns+1) = (100.0/rows) * i
      enddo

      !set top to 0 and bottom to linear increase
      do j=0,columns+1
         A(0,j) = 0.0
         A(rows+1,j) = ((100.0)/columns) * j
      enddo

end subroutine initialize

!print diagonal in bottom corner where most action is
subroutine track_progress(Anew, iteration)
      implicit none

      integer, parameter              :: columns=1000
      integer, parameter              :: rows=1000
      integer                         :: i,iteration

      double precision, dimension(0:rows+1,0:columns+1) :: Anew

      print *, '---------- Iteration number: ', iteration, ' ---------------'
      do i=5,0,-1
         write (*,'("("i4,",",i4,"):",f6.2,"  ")',advance='no'), &
                     rows-i,columns-i,Anew(rows-i,columns-i)
      enddo
      print *
end subroutine track_progress
```

OPENACC SYNTAX

OPENACC SYNTAX

Syntax for using OpenACC directives in code

• C/C++	• Fortran
#pragma acc *directive clauses* *<code>*	!$acc *directive clauses* *<code>*

- 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.

General Directive Syntax and Scope

Fortran

```
!$acc kernels [clause …]
    structured block
!$acc end kernels
```

C

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

I may indent the directives at the natural code indentation level for readability.

It is a common practice to always start them in the first column (ala #define/#ifdef).
Either is fine with C or Fortran 90 compilers.

A Simple Example: SAXPY

SAXPY in 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];
}

...
// Somewhere in main
// call SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```

SAXPY in 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

...
$ From main program
$ call SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...
```

kernels: Our first OpenACC Directive

We request that each loop execute as a separate *kernel* on the GPU. This is an incredibly powerful directive.

```
!$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 1

kernel 2

Kernel:
A parallel routine to run on the parallel hardware

Complete SAXPY Example Code

```
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;
}
```

```
#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];
}
```

"I promise y is not aliased by Anything else (esp. x)"

Compile and Run

C: `$nvc -acc saxpy.c`

Fortran: `$nvfortran -acc saxpy.f90`

Run: `$./a.out`

-acc will enable OpenACC directives
Default here targets serial CPU and GPU

-ta=tesla will *only* target a GPU
-ta=multicore will only target a multicore CPU
-Minfo=accel turns on helpful compiler reporting

```
$ nvc -acc -Minfo=accel -ta=tesla 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
```

C Detail: the “restrict” keyword

- Standard C (as of C99).
- Important for optimization of serial as well as OpenACC and OpenMP code.
- Promise given by the programmer to the compiler for a pointer:

  float *restrict ptr

  Meaning: “for the lifetime of ptr, only it or a value directly derived from it (such as ptr + 1) will be used to access the object to which it points”
- Limits the effects of pointer aliasing
- OpenACC compilers often require restrict to determine independence
  - Otherwise the compiler can’t parallelize loops that access ptr
  - Note: if programmer violates the declaration, behavior is undefined

Compare: OpenACC and CUDA Implementations

OpenACC: Complete SAXPY Example Code

```
#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;
}
```

CUDA: Partial CUDA C SAXPY Code SAXPY Example Code

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

cudaMemcpyHostToDevice );
  cudaMemcpy( yd, y, n*sizeof(float),

cudaMemcpyHostToDevice );
  saxpy_kernel<<< (n+31)/32, 32 >>>( a,
xd, yd, n );
  cudaMemcpy( x, xd, n*sizeof(float),

cudaMemcpyDeviceToHost );
  cudaFree( xd ); cudaFree( yd );
}
```

```
module kmod
 use cudafor
contains
 attributes(global) subroutine
    saxpy_kernel(A,X,Y,N)
  real(4), device :: A, X(N), Y(N)
  integer, value :: N
  integer :: i
  i = (blockidx%x-1)*blockdim%x + threadidx%x
  if( i <= N ) X(i) = A*X(i) + Y(i)
 end subroutine
end module


subroutine saxpy( A, X, Y, N )
  use kmod
  real(4) :: A, X(N), Y(N)
  integer :: N
  real(4), device, allocatable,
    dimension(:):: &
    Xd, Yd
  allocate( Xd(N), Yd(N) )
  Xd = X(1:N)
  Yd = Y(1:N)
  call saxpy_kernel<<<(N+31)/32,32>>>(A, Xd,
    Yd, N)
  X(1:N) = Xd
  deallocate( Xd, Yd )
end subroutine
```

Big Difference!

OpenACC vs CUDA implementations

- **CUDA: Hard to Maintain, OpenACC: Easy to Maintain**
  - With CUDA, we changed the structure of the old code. Non-CUDA programmers can't understand new code. It is not even ANSI standard code.
- **CUDA: Rewrite Original Code, OpenACC: Augment Original Code**
  - We have separate sections for the host code and the GPU code. Different flow of code. Serial path now gone forever.
- **CUDA: Optimized for Specific Hardware, OpenACC: One Source Everywhere**
  - Where did these "32"s and other mystery numbers come from? This is a clue that we have some hardware details to deal with here.
- **CUDA: Assembler-like Programming, OpenACC: Relies on Compiler**
  - Exact same situation as assembly used to be. How much hand-assembled code is still being written in HPC now that compilers have gotten so efficient?

This looks easy! Too easy…

Questions:

1. If it is this simple, why don't we just throw *kernels* in front of every loop?
2. Better yet, why doesn't the compiler do this for me?

Answers:

There are two general issues that prevent the compiler from being able to just automatically parallelize every loop:

1. Data Dependencies in Loops
2. Data Movement

The compiler needs your higher level perspective (in the form of directive hints) to get correct results and reasonable performance

DATA Dependencies

Data Dependencies

Most directive based parallelization consists of splitting up big do/for loops into independent chunks that the many processors can work on simultaneously.

Take, for example, a simple for loop like this:

```
for(index=0, index<1000000,index++)
    Array[index] = 4 * Array[index];
```

When run on 1000 processors, it will execute something like this…

No Data Dependencies

A run on 1000 processors for the loop below

```
for(index=0, index<1000000,index++)
    Array[index] = 4 * Array[index];
```

Processor 1

```
for(index=0, index<999,index++)
    Array[index] = 4*Array[index];
```

Processor 2

```
for(index=1000, index<1999,index++)
    Array[index] = 4*Array[index];
```

Processor 3

```
for(index=2000, index<2999,index++)
    Array[index] = 4*Array[index];
```

Processor 4

```
for(index=3000, index<3999,index++)
    Array[index] = 4*Array[index];
```

Processor 5

```
for(index=4000, index<4999,index++)
    Array[index] = 4*Array[index];
```

....

With Data Dependencies

But what if the loops are not entirely independent?

Take, for example, a similar loop like this:

```
for(index=1, index<1000000,index++)
    Array[index] = 4 * Array[index] - Array[index-1];
```

Added data dependency

This is a perfectly valid serial code.

WITH Data Dependencies

Processor 1

```
for(index=0, index<999,index++)
   Array[index] = 4*Array[index]-
Array[index-1];
```

Processor 2

```
for(index=1000, index<1999,index++)
   Array[index] = 4*Array[index]-
Array[index-1];
```

. . . .

Result from Processor 1

```
for(index=1000, index<1999,index++)
        Array[1000] = 4 * Array[1000] – Array[999];
```

Needs the result of Processor 1's last iteration.

If we want the correct ("same as serial") result, we need to wait until processor 1 finishes. Likewise for processors 3, 4, …

Data Dependencies

If the compiler even *suspects* that there is a data dependency, it will, for the sake of correctness, refuse to parallelize that loop.

```
11, Loop carried dependence of 'Array' prevents parallelization
    Loop carried backward dependence of 'Array' prevents vectorization
```

As large, complex loops are quite common in HPC, especially around the most important parts of your code, the compiler will often balk most when you most need a kernel to be generated. What can you do?

How To Manage Data Dependencies

- Rearrange your code to make it more obvious to the compiler that there is not really a data dependency.
- Eliminate a real dependency by changing your code.
  - There is a common bag of tricks developed for this as this issue goes back 40 years in HPC. Many are quite trivial to apply.
  - The compilers have gradually been learning these themselves.
- Override the compiler's judgment (`independent` clause) at the risk of invalid results. Misuse of `restrict` has similar consequences.

Till we discussed

- What OpenACC is used for
- What a Directive is
- The OpenACC kernels directive
- What a dependency is
- How to compile an OpenACC enabled code

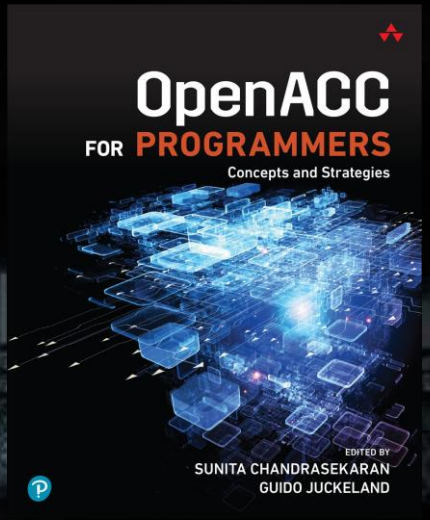
Cloud Innovator MEGAZONE

OpenACC
More Science, Less Programming

BOOK: OpenACC for Programmers

Edited by: By Sunita Chandrasekaran, Guido Juckeland

- Discover how OpenACC makes scalable parallel programming easier and more practical
- Get productive with OpenACC code editors, compilers, debuggers, and performance analysis tools
- Build your first real-world OpenACC programs
- Overcome common performance, portability, and interoperability challenges
- Efficiently distribute tasks across multiple processors



READ MORE

Use code **OPENACC** during checkout for 35% off

Pearson
Addison-Wesley

**Discount taken off list price. Offer only good at informit.com*

Cloud Innovator MEGAZONE

OpenACC
More Science, Less Programming

OPENACC Resources

Guides • Talks • Tutorials • Videos • Books • Spec • Code Samples • Teaching Materials • Events • Success Stories • Courses • Slack • Stack Overflow

FREE Compilers

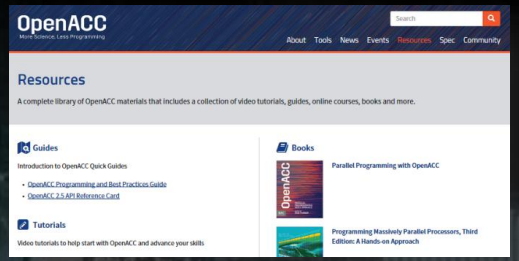
PGI Community EDITION

slack

https://www.openacc.org/community#slack

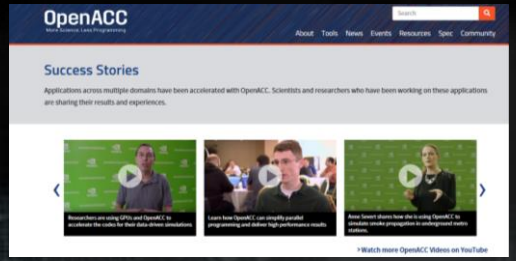
Resources

https://www.openacc.org/resources



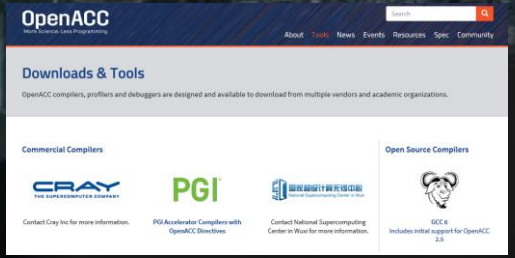
Success Stories

https://www.openacc.org/success-stories



Compilers and Tools

https://www.openacc.org/tools



Events

https://www.openacc.org/events

OpenACC Directives

https://www.openacc.org/sites/default/files/inline-files/openacc-guide.pdf

OpenACC Courses

- 3 Part Introduction to OpenACC
- Module 1 – Introduction to OpenACC
- Module 2 – Data Management with OpenACC
- Module 3 – Optimizations with OpenACC
- Each module will have a corresponding lab

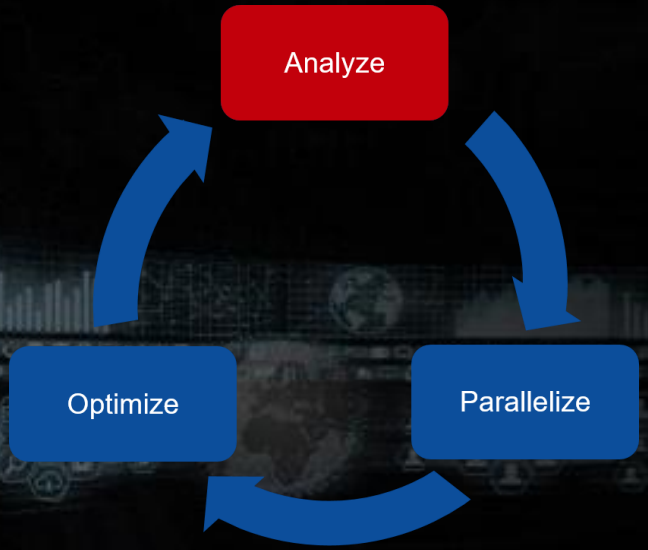
MODULE 2 OUTLINE

Topics to be covered

- CPU and GPU Memories
- CUDA Unified (Managed) Memory
- OpenACC Data Management
- Lab 2

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.



Cloud Innovator
MEGAZONE

Megazone is delivering values through the business expertise and experiences built from the Platform Service, Digital Marketing and Digital Service areas

OpenACC
More Science, Less Programming

OPENACC PARALLEL LOOP DIRECTIVE

OpenACC Directives

Manage Data Movement

Initiate Parallel Execution

Optimize Loop Mappings

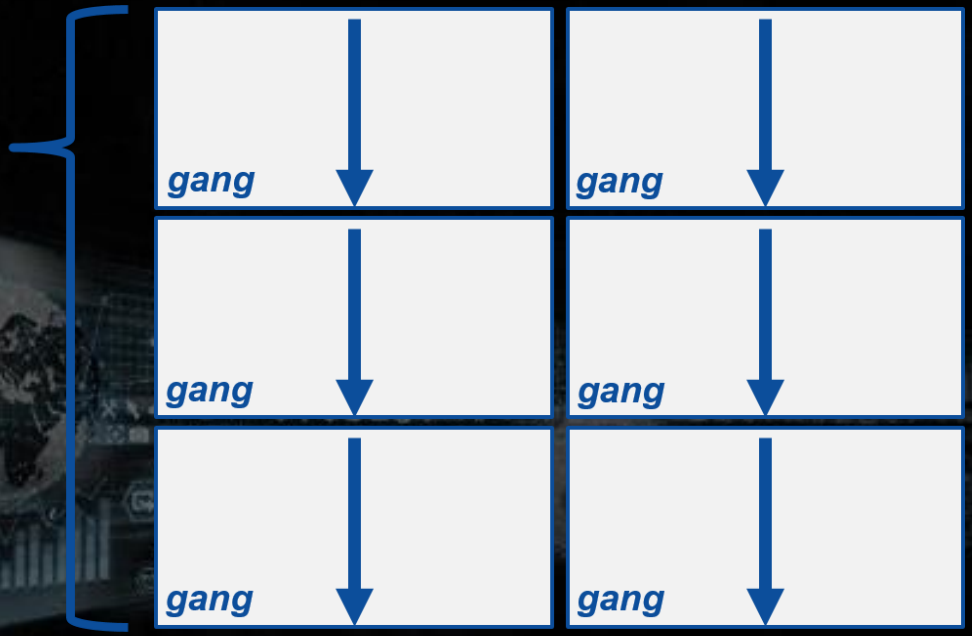
```
#pragma acc data copyin(a,b) copyout(c)
{
#pragma acc parallel
{
    #pragma acc loop gang
        for (i = 0; i < n; ++i) {
    #pragma acc loop vector
            for (j = 0; j < n; ++j) {
                c[i][j] = a[i][j] + b[i][j];
...
```

- Incremental
- Single source
- Interoperable
- Performance portable
- CPU, GPU, Manycore

OPENACC PARALLEL DIRECTIVE

Expressing parallelism

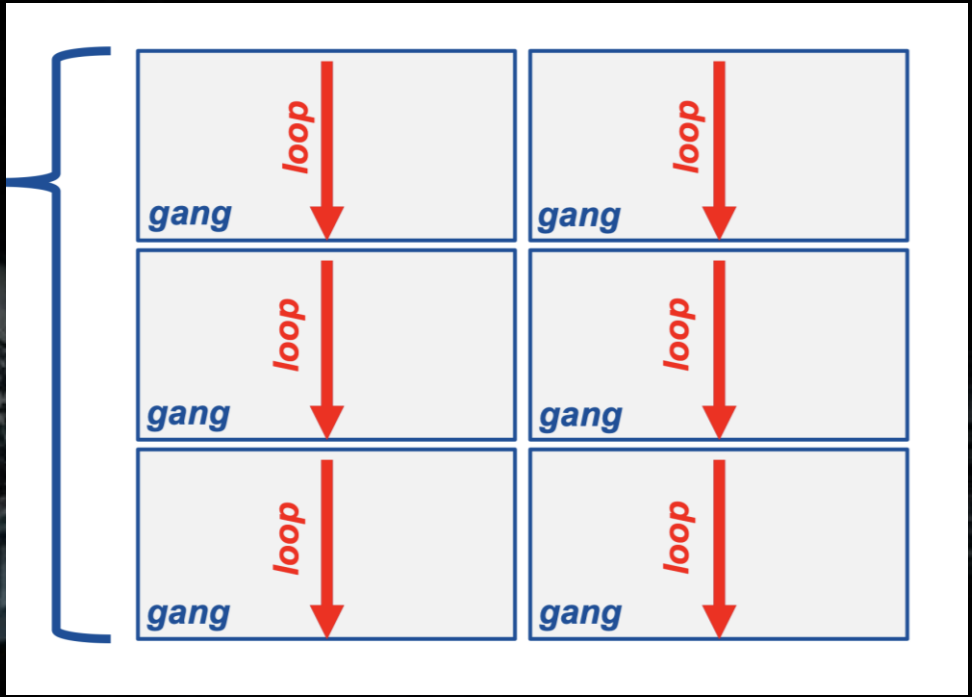
```
#pragma acc parallel
{
    When encountering the parallel directive,
    the compiler will generate 1 or more
    parallel gangs, which execute redundantly.
}
```



OPENACC PARALLEL DIRECTIVE

Expressing parallelism

```
#pragma acc parallel
{
    for (int i=0; i<N; i++)
    {
    //Do something
    }
    //This loop will be executed redundantly parallelized across the gangs
    // This means that each gang will execute the entire loop.

}
```



OPENACC PARALLEL LOOP DIRECTIVE

Parallelizing many loops

```
#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;
```

- To parallelize multiple loop nests, each should be accompanied by a parallel directive
- Each parallel loop nest can have different loop boundaries and loop optimizations
- Each parallel loop nest can be parallelized in a different way
- This is the recommended way to parallelize multiple loop nests.
  - Attempting to parallelize multiple loop nests within the same parallel region may give performance issues or unexpected results.

PARALLELIZE WITH OPENACC PARALLEL LOOP

```
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 net, max reduction required

Parallelize second loop

We didn’t detail how to parallelize the loops, just which loops to parallelize.

OpenACC Directives

Manage Data Movement

Initiate Parallel Execution

Optimize Loop Mappings

```
#pragma acc data copyin(a,b) copyout(c)
{
#pragma acc parallel
{
    #pragma acc loop gang
        for (i = 0; i < n; ++i) {
    #pragma acc loop vector
            for (j = 0; j < n; ++j) {
                c[i][j] = a[i][j] + b[i][j];
...
```

- Incremental
- Single source
- Interoperable
- Performance portable
- CPU, GPU, Manycore

REDUCTION CLAUSE

```
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];
```

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

- The **reduction** clause takes many values and "***reduces***" them to a single value, such as in a sum or maximum
- Each partial result is calculated in parallel
- A ***single result*** is created by combining the partial results using the specified operation

REDUCTION CLAUSE OPERATORS

Operator	Description	Example
+	Addition/Summation	reduction(+:sum)
*	Multiplication/Product	reduction(*:product)
max	Maximum value	reduction(max:maximum)
min	Minimum value	reduction(min:minimum)
&	Bitwise and	reduction(&:val)
|	Bitwise or	reduction(|:val)
&&	Logical and	reduction(&&:val)
||	Logical or	reduction(||:val)

BUILD AND RUN THE CODE

PGI & NV COMPILER BASICS

pgcc/nvcc, pgc++/nvc++ and pgfortran/nvfortran

- The command to compile C code is ‘pgcc’ or ‘nvcc’
- The command to compile C++ code is ‘pgc++’ or ‘nvc++
- The command to compile Fortran code is ‘pgfortran’ or ‘nvfortran’
- The -fast flag instructs the compiler to optimize the code to the best of its abilities

```
$ pgcc –fast main.c
$ pgc++ -fast main.cpp
$ pgfortran –fast main.F90
```

PGI & NV COMPILER BASICS

–Minfo flag

- The -Minfo flag will instruct the compiler to print feedback about the compiled code
- -Minfo=accel will give us information about what parts of the code were accelerated via OpenACC
- -Minfo=opt will give information about all code optimizations
- -Minfo=all will give all code feedback, whether positive or negative

$ pgcc –fast –Minfo-=all main.c → nvc

$ pgc++ -fast –Minfo=all main.cpp → nvc++

$ nvfortran –fast –Minfo=all main.F90 → nvfortran

PGI & NV COMPILER BASICS

–ta flag

- The -ta flag enables building OpenACC code for a "Target Accelerator" (TA)
- -ta=multicore – Build the code to run across threads on a multicore CPU
- -ta=tesla:managed – Build the code for an NVIDIA (Tesla) GPU and manage the data movement automatically (more next module)

```
$ pgcc –fast –Minfo-=all –ta=tesla:managed main.c        → nvc
$ pgc++ -fast –Minfo=all –ta=tesla:managed main.cpp      → nvc++
$ nvfortran –fast –Minfo=all –ta=tesla:managed main.F90  → nvfortran
```

Exercises: General Instructions

- The exercise is found at https://github.com/kesperinc/NERSC_OPENACC.git
- Your objective is to add OpenACC to the serial C or Fortran code to enable it to parallelize
- Hint: look for the significant loops and apply the one tool you have

Lab1 Exercise

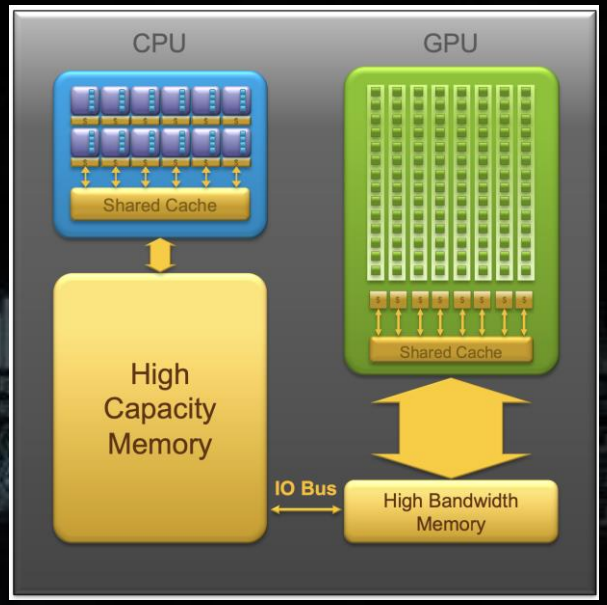
- Check the elapsed time of the serial and the code with -fast options
- Compare the time with/without options

CPU and GPU Memories

CUDA UNIFIED MEMORY

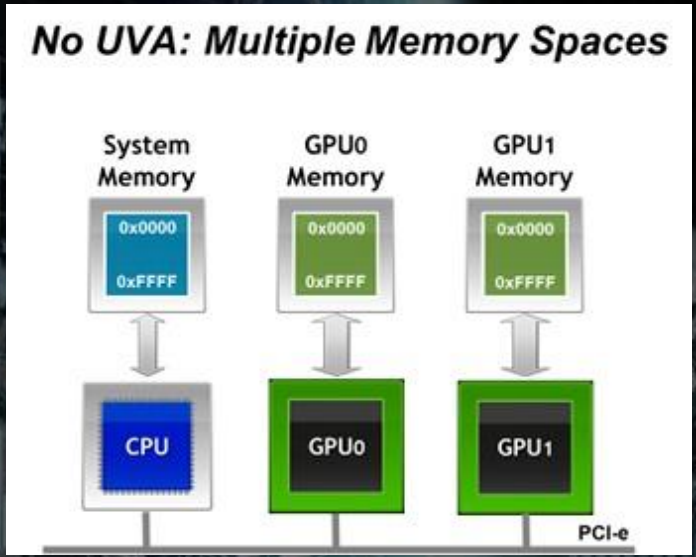
CPU and GPU

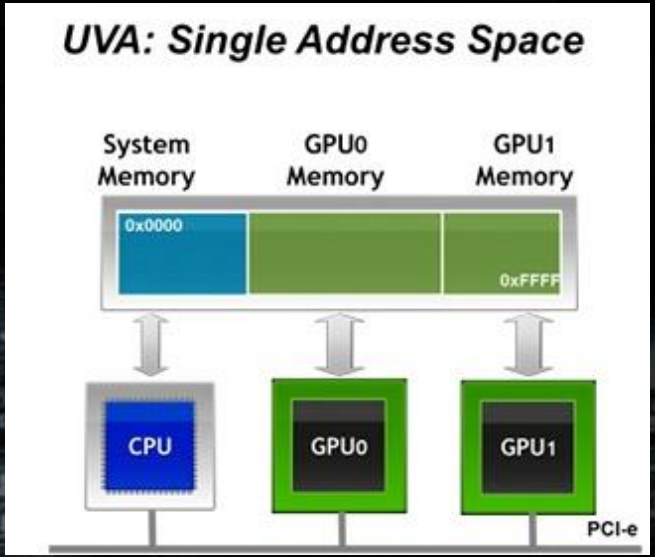
- CPU memory is larger, GPU memory has more bandwidth
- CPU and GPU memory are usually separate, connected by an I/O bus (traditionally PCIe)
- Any data transferred between the CPU and GPU will be handled by the I/O Bus
- The I/O Bus is relatively slow compared to memory bandwidth
- The GPU cannot perform computation until the data is within its memory



CUDA Unified Memory

Commonly referred to as "managed memory."





PCIe or NVLink

CPU and GPU memories are combined into a single, shared pool

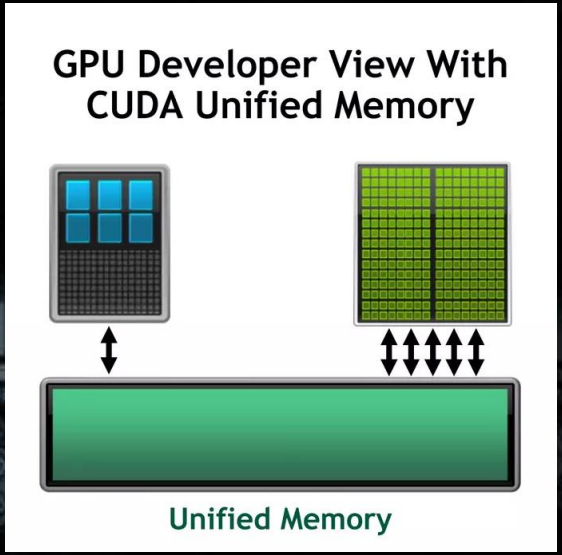
CUDA Unified Memory

- Handling explicit data transfers between the host and device (CPU and GPU) can be difficult
- The PGI compiler can utilize CUDA Managed Memory to defer data management
- This allows the developer to concentrate on parallelism and think about data movement as an optimization

```
$ pgcc –fast –ta=tesla:managed –Minfo=accel main.c
$ pgfortran –fast –ta=tesla:managed –Minfo=accel main.f90
```

Limitations of Managed Memory

- The programmer will almost always be able to get better performance by manually handling data transfers
- Memory allocation/deallocation takes longer with managed memory
- Cannot transfer data asynchronously
- Currently only available from PGI on NVIDIA GPUs.

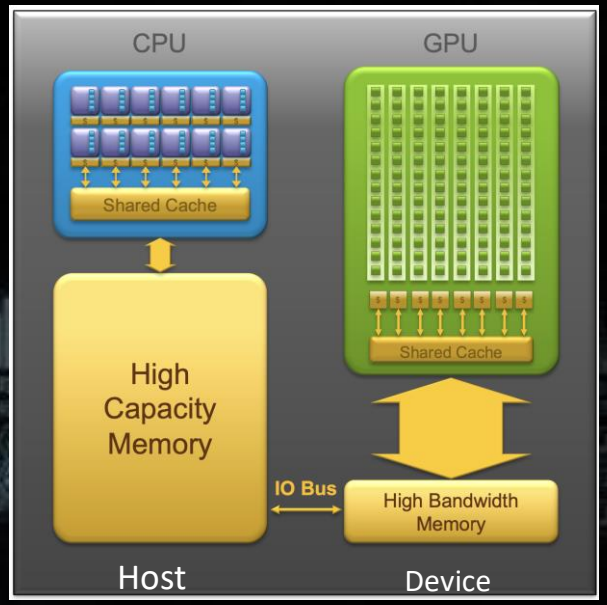


C malloc, C++ new, Fortran allocate all managed memory to CUDA Unified Memory

BASIC DATA MANAGEMENT

Between the host and device

- The host is traditionally a CPU
- The device is some parallel accelerator
- When our target hardware is multicore, the host and device are the same, meaning that their memory is also the same
  - There is no need to explicitly manage data when using a shared memory accelerator, such as the multicore target
- When the target hardware is a GPU, data will usually need to migrate between CPU and GPU memory
  - Each array used on the GPU must be allocated on the GPU
  - When data changes on the CPU or GPU the other must be updated



Cloud Innovator
MEGAZONE

Megazone is delivering values through the business expertise and experiences built from the Platform Service, Digital Marketing and Digital Service areas

OpenACC
More Science, Less Programming

Data Shaping

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

```
copy( array[starting_index:length] )          : C/C++
copy( array(starting_index:ending_index) )    : Fortran
```

Array Shaping (cont.)

Multidimension & Partial Array Shaping

These examples copy a 2D array to the device

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

copy(array(1:N, 1:M)) : Fortran

These examples copy only ¼ of the full array

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

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

Optimized Data Movement

```
while ( err > tol && iter < iter_max ) {
        err=0.0;
#pragma acc parallel loop reduction(max:err)  copyin(A[0:n*m])  copy(Anew[0:n*m])
 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 copyin(Anew[0:n*m]) copyout(A[0:n*m])
 for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
                A[j][i] = Anew[j][i];
        }
}
 iter++;
}
```

Data clauses provide necessary "shape" of the arrays.

OpenACC Data Directive

Definition

- The data directive defines a lifetime for data on the device beyond individual loops
- During the region data is essentially "owned by" the accelerator
- Data clauses express shape and data movement for the region

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

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

STRUCTURED DATA DIRECTIVE

Example

```
#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];
        }
}
```

Optimized Data Movement

```
#pragma acc data copyin(A[0:n*m])  copy(Anew[0:n*m])
while ( err > tol && iter < iter_max ) {
        err=0.0;
#pragma acc parallel loop reduction(max:err) copyin(A[0:n*m]) copy(Anew[0:n*m])
  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 copyin(Anew[0:n*m]) copyout(A[0:n*m])
  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

Cloud Innovator
MEGAZONE

Megazone is delivering values through the business expertise and experiences built from the Platform Service, Digital Marketing and Digital Service areas

OpenACC
More Science, Less Programming

Lab Exercise with Data Directives

WHAT WE'VE LEARNED SO FAR

- CUDA Unified (Managed) Memory is a powerful porting tool
- GPU programming without managed memory often requires data shaping
- Moving data at each loop is often inefficient
- The OpenACC Data region can decouple data movement and computation

Cloud Innovator
MEGAZONE

Megazone is delivering values through the business expertise and experiences built from the Platform Service, Digital Marketing and Digital Service areas

OpenACC
More Science, Less Programming

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

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

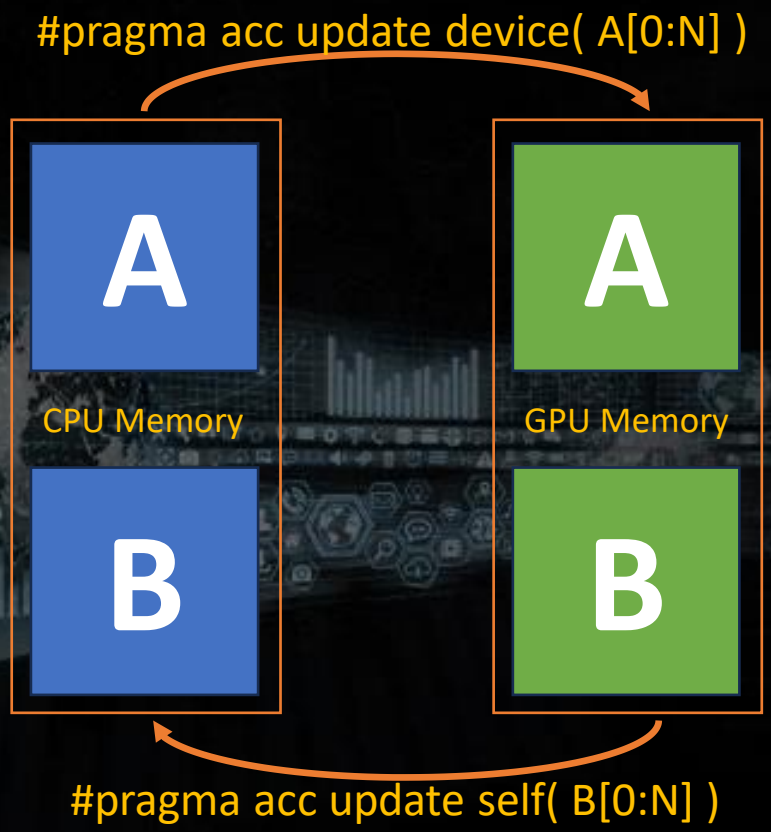
: C/C++

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



SYNCHRONIZE DATA WITH UPDATE

```
int* A=(int*) malloc(N*sizeof(int)

#pragma acc data create(A[0:N])
while( timesteps++ < numSteps )
{
        #pragma acc parallel loop
        for(int i = 0; i < N; i++){
                a[i] *= 2;
        }
        if (timestep % 100 ){
        #pragma acc update self(A[0:N])
        checkpointAToFile(A, N);
        }
}
```

- Sometimes data changes on the host or device inside a data region
- Ending the data region and starting a new one is expensive
- Instead, update the data so that the host and device data are the same
- Examples: File I/O, Communication, etc.

UNSTRUCTURED DATA DIRECTIVES

UNSTRUCTURED DATA DIRECTIVES

Enter/Exit Data Directive

- Data lifetimes aren't always neatly structured.
- The *enter data* directive handles device memory allocation
- You may use either the create or the copyin clause for memory allocation
- The *enter data* directive is **not** the start of a data region, because you may have multiple *enter data* directives

```
#pragma acc enter data clauses
        < Sequential and/or Parallel Code>
#pragma acc exit data clauses
```

- The exit data directive handles device memory deallocation
- You may use either the delete or the copyout clause for memory deallocation
- You should have as many exit data for a given array as enter data
- These can exist in different functions

```
!$acc enter data clauses
        < Sequential and/or Parallel Code>
!$acc exit data clauses
```

UNSTRUCTURED DATA CLAUSES

enter data:

- copyin (list) Allocates memory on device and copies data from host to device on enter data.
- create (list) Allocates memory on device without data transfer on enter data.

exit data:

- copyout (list) Allocates memory on device and copies data back to the host on exit data.
- delete (list) Deallocates memory on device without data transfer on exit data.

UNSTRUCTURED DATA DIRECTIVES

Basic Example

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

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

UNSTRUCTURED DATA DIRECTIVES

With a simple code

Unstructured

- Can have multiple starting/ending points
- Can branch across multiple functions
- Memory exists until explicitly deallocated

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

Structured

- Must have explicit start/end points
- Must be within a single function
- Memory only exists within the data region

```
#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];
        }
}
```

C++ STRUCTS/CLASSES

With dynamic data members

- C++ Structs/Classes work the same exact way as they do in C
- The main difference is that now we have to account for the implicit “this” pointer

```
class vector {
        private:
          float *arr;
          int n;
        public:
          vector(int size){
                n = size;
                arr = new float[n];
                #pragma acc enter data copyin(this)
                #pragma acc enter data create(arr[0:n])
        }
        ~vector(){
          #pragma acc exit data delete(arr)
          #pragma acc exit data delete(this)
          delete(arr);
        }
};
```

UNSTRUCTURED DATA DIRECTIVES

Branching across multiple functions

- In this example enter data and exit data are in different functions
- This allows the programmer to put device allocation/deallocation with the matching host versions
- This pattern is particularly useful in C++, where structured scopes may not be possible.

```
int* allocate_array(int N){
        int* ptr = (int *) malloc(N * sizeof(int));
        #pragma acc enter data create(ptr[0:N])
        return ptr;
}
void deallocate_array(int* ptr){
        #pragma acc exit data delete(ptr)
        free(ptr);
}
int main(){
        int* a = allocate_array(100);
        #pragma acc kernels
        {
                a[0] = 0;
        }
        deallocate_array(a);
}
```

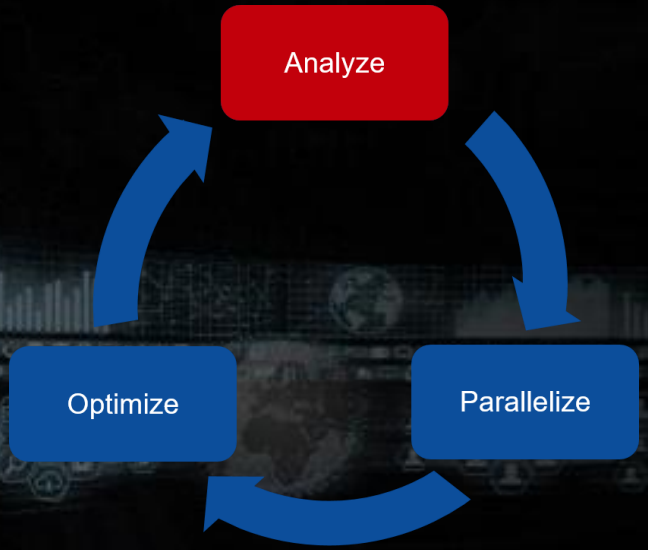
KEY CONCEPTS

- Differences between CPU, GPU, and Unified Memories
- OpenACC Array Shaping
- OpenACC Data Clauses
- OpenACC Structured Data Region
- OpenACC Update Directive
- OpenACC Unstructured Data Directives

Next: Loop Optimizations

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.



OpenACC Directives

Manage Data Movement

Initiate Parallel Execution

Optimize Loop Mappings

```
#pragma acc data copyin(a,b) copyout(c)
{
#pragma acc parallel
{
    #pragma acc loop gang
        for (i = 0; i < n; ++i) {
    #pragma acc loop vector
            for (j = 0; j < n; ++j) {
                c[i][j] = a[i][j] + b[i][j];
...
```

- Incremental
- Single source
- Interoperable
- Performance portable
- CPU, GPU, Manycore

PARALLELIZE WITH OPENACC PARALLEL LOOP

```
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 net, max reduction required

Parallelize second loop

We didn’t detail how to parallelize the loops, just which loops to parallelize.

Optimized Data Movement

```
#pragma acc data copyin(A[0:n*m])  copy(Anew[0:n*m])
while ( err > tol && iter < iter_max ) {
        err=0.0;
#pragma acc parallel loop reduction(max:err) copyin(A[0:n*m]) copy(Anew[0:n*m])
  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 copyin(Anew[0:n*m]) copyout(A[0:n*m])
  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

Cloud Innovator
MEGAZONE

Megazone is delivering values through the business expertise and experiences built from the Platform Service, Digital Marketing and Digital Service areas

OpenACC
More Science, Less Programming

GANGS, WORKERS, AND VECTORS DEMYSTIFIED

Cloud Innovator
MEGAZONE

OpenACC
More Science, Less Programming

GANGS, WORKERS, AND VECTORS DEMYSTIFIED

GANGS, WORKERS, AND VECTORS DEMYSTIFIED

- How much work 1 worker can do is limited by his speed.
- A single worker can only move so fast.
- Even if we increase the size of his roller, he can only paint so fast.
- We need more workers!

GANGS, WORKERS, AND VECTORS DEMYSTIFIED

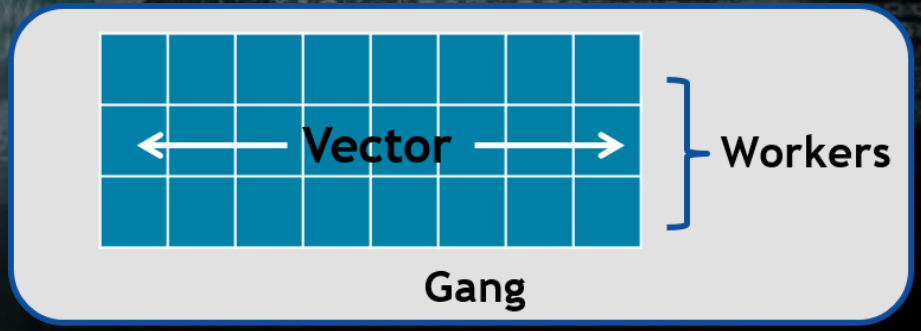
- Multiple workers can do more work and share resources, if organized properly.

GANGS, WORKERS, AND VECTORS DEMYSTIFIED

- By organizing our workers into groups (gangs), they can effectively work together within a floor.
- Groups (gangs) on different floors can operate independently.
- Since gangs operate independently, we can use as many or few as we need.
- Even if there’s not enough gangs for each floor, they can move to another floor when ready.

GANGS, WORKERS, AND VECTORS DEMYSTIFIED

- Our painter is like an OpenACC **worker**, he can only do so much.
- His roller is like a **vector**, he can move faster by covering more wall at once.
- Eventually we need more workers, which can be organized into **gangs** to get more done.



LOOP OPTIMIZATIONS

OPENACC LOOP DIRECTIVE

Expressing parallelism

- Mark a single for loop for parallelization
- Allows the programmer to give additional information and/or optimizations about the loop
- Provides many different ways to describe the type of parallelism to apply to the loop
- Must be contained within an OpenACC compute region (either a kernels or a parallel region) to parallelize loops

```
#pragma acc loop
for  (int i = 0; i < N ; i++)
// Do something
```

```
!$acc loop
Do i = 1, N
! Do something
```

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

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

COLLAPSE CLAUSE

```
#pragma acc data copy(A[0:n*m])  copyin(Anew[0:n*m])
while ( err > tol && iter < iter_max ) {
        err=0.0;
#pragma acc parallel loop reduction(max:err) collapse(2) copyin(A[0:n*m]) copy(Anew[0:n*m])
 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 collapse(2) copyin(Anew[0:n*m]) copyout(A[0:n*m])
 for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
                A[j][i] = Anew[j][i];
        }
}
 iter++;
}
```

Collapse 2 loops into one
for more flexibility in parallelizing

TILE CLAUSE

- tile (x, y, z, …)

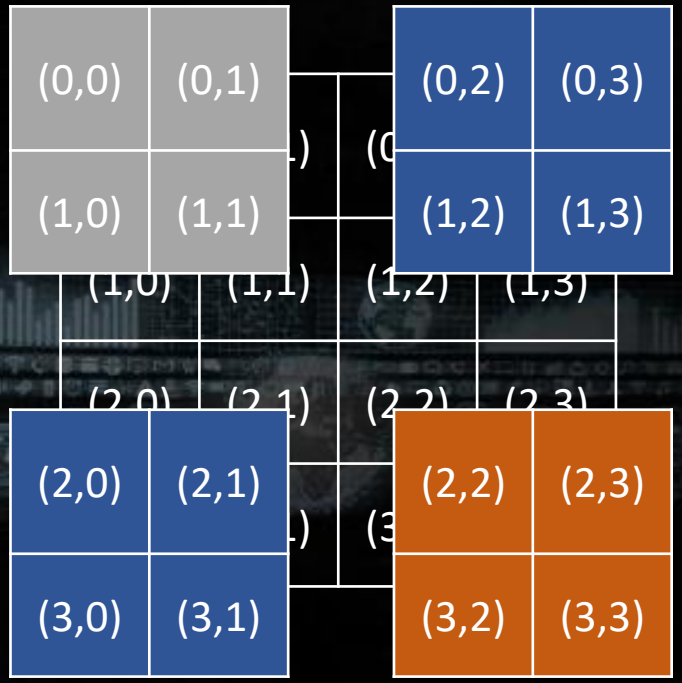
- Breaks multidimensional loops into “tiles” or “blocks”
- Can increase data locality in some codes
- Will be able to execute multiple “tiles” simultaneously

```
#pragma acc kernels loop tile(32,32)
for  (int i = 0; i < size ; i++ )
  for ( int j = 0; j < size ; j++ )
    for ( int j = 0; j < size ; j++ )
                c[i][j] += a[i][j]*b[k][j];
```

TILE CLAUSE

```
#pragma acc kernels loop tile(2,2)
for  (int i = 0; i < 4 ; i++ )
  for ( int j = 0; j < 4 ; j++ )
            array[i][j]++;
```

tile(2, 2)



TILE CLAUSE

```
#pragma acc data copy(A[0:n*m]) copyin(Anew[0:n*m])
while ( err > tol && iter < iter_max ) {
        err=0.0;
#pragma acc parallel loop reduction(max:err) tile(32,32) copyin(A[0:n*m]) copy(Anew[0:n*m])
 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) copyin(Anew[0:n*m]) copyout(A[0:n*m])
 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

TILING Results (V100)

- The collapse clause often requires an exhaustive search of options.
- For our example code...
  - CPU saw no benefit from tiling
  - GPU saw anywhere from a 15% loss of performance to a 25% improvement

	CPU	GPU
baseline	1.00x	1.00x
4x4	1.00x	0.85x
4x8	1.00x	0.95x
8x4	1.00x	0.99x
8x8	1.00x	1.03x
16x8	1.00x	1.09x
16x16	1.00x	1.11x
16x32	1.00x	1.18x
32x16	1.00x	1.22x
32x32	1.00x	1.25x

GANG, WORKER, AND VECTOR CLAUSES

- The developer can instruct the compiler which levels of parallelism to use on given loops by adding clauses:
  - gang – Mark this loop for gang parallelism
  - worker – Mark this loop for worker parallelism
  - vector – Mark this loop for vector parallelism
- These can be combined on the same loop.

```
#pragma acc parallel loop gang
for( i = 0; i < size; i++ )
  #pragma acc loop worker
  for( j = 0; j < size; j++ )
    #pragma acc loop vector
    for( k = 0; k < size; k++ )
      c[i][j] += a[i][k] * b[k][j];
```

```
#pragma acc parallel loop \
                collapse(3) gang vector
for( i = 0; i < size; i++ )
  #pragma acc loop worker
  for( j = 0; j < size; j++ )
    #pragma acc loop vector
    for( k = 0; k < size; k++ )
      c[i][j] += a[i][k] * b[k][j];
```

SEQ CLAUSE

- The seq clause (short for sequential) will tell the compiler to run the loop sequentially
- In the sample code, the compiler will parallelize the outer loops across the parallel threads, but each thread will run the inner-most loop sequentially
- The compiler may automatically apply the seq clause to loops as well

```
#pragma acc parallel loop
for  (int i = 0; i < size ; i++ )
  #pragma acc loop
  for ( int j = 0; j < size ; j++ )
    #pragma acc loop seq
    for ( int k = 0; k < size ; k++ )
            c[i][j] += a[i][k] * b[k][j];
```

ADJUSTING GANGS, WORKERS AND VECTORS

The compiler will choose a number of gangs, workers, and a vector length for you, but you can change it with clauses.

- num_gangs(N) : Generate N gangs for this parallel region
- num_workers(M) : Generate M workers for this parallel region
- vector_length(Q) : Use a vector length of Q for this parallel region

```
#pragma acc parallel num_gangs(2) \
   num_workers(2) vector_length(32)
{
  #pragma acc loop gang worker
  for( int i = 0; i < 4; i++ ){
    #pragma acc loop vector
    for(int j = 0; j < 32; j++ ){
                array[i][j] ++;
    }
  }
}
```

COLLAPSE CLAUSE with Vector Length

```
#pragma acc data copy(A[0:n*m])  copyin(Anew[0:n*m])
while ( err > tol && iter < iter_max ) {
        err=0.0;
#pragma acc parallel loop reduction(max:err) collapse(2) vector_length(1024) \
        copyin(A[0:n*m]) copy(Anew[0:n*m])
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 collapse(2) vector_length(1024) \
        copyin(Anew[0:n*m]) copyout(A[0:n*m])
for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
                A[j][i] = Anew[j][i];
        }
}
iter++;
}
```

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 on NVIDIA GPUs. When the vector length is very short, a worker loop can increase the parallelism in your gang (thread block).
- When possible, the vector loop should step through your arrays consecutively (stride==1)
- Gangs should come from outer loops, vectors from inner

KEY CONCEPTS

- Some details that are available to use from a GPU profile
- Gangs, Workers, and Vectors Demystified
- Collapse clause
- Tile clause
- Gang/Worker/Vector clauses

OpenACC directive syntax

	sentinel	construct	clauses	
C/C++	#pragma acc	data	copy(data)	data model
		update	host(data)	data model
		kernels		execution model
		parallel		execution model
Fortran	!$acc	data	copy(data)	data model
		update	host(data)	data model
		kernels		execution model
		parallel		execution model

OpenACC directive syntax

- OpenACC uses compiler directives for defining compute regions (and data transfers) that are to be performed on a GPU
- Important constructs
  - parallel, kernels, data, loop, update, host_data, wait
- Often used clauses
  - if (condition), async(handle)

End of OpenACC Tutorials