CIS 631
Advanced Parallel Computing

Parallel Programming Methods:
GPU (OpenACC)

Prof. Allen D. Malony
Department of Computer and Information Science
Winter 2022
Logistics

- All videos should now be available soon
- OpenMP exercise due on Sunday, midnight
- Term paper (see link on CIS 631 webpage)
  - Abstract and references due February 14
  - Comparative summary of research work on topic of interest to you
What is OpenACC?

- OpenACC is a directives-based programming approach to parallel computing
  - Designed for performance and portability
  - Runs on CPUs and accelerators for HPC
- Directives used to specify parallelism
  - Incremental
  - Single source
  - Low learning curve
# History of OpenACC

<table>
<thead>
<tr>
<th>Year</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>OpenACC 1.0 specification is released at SC11 (NVIDIA, Cray, PGI, CAPS)</td>
</tr>
<tr>
<td>2013</td>
<td>OpenACC 2.0: More functionality, portability</td>
</tr>
<tr>
<td>2015</td>
<td>OpenACC 2.5: Enhancements, clarifications</td>
</tr>
<tr>
<td>2017</td>
<td>OpenACC 2.6: Deep copy, …</td>
</tr>
<tr>
<td>2019</td>
<td>OpenACC 3.0: Newer C++, more lambdas, …</td>
</tr>
<tr>
<td>2020</td>
<td>OpenACC 3.1: C++ range-based for loops, Fortran DO CONCURRENT, …</td>
</tr>
</tbody>
</table>

- Run as a non-profit organization, OpenACC.org
- Members from industry and academia

→ [https://www.openacc.org/](https://www.openacc.org/) (see also: Best practice guide)

OpenACC-enabled Applications
- ANSYS Fluent
- Gaussian
- VASP
- COSMO
- GTC
- SOMA
- …
OpenACC and OpenMP

- OpenACC modeled after OpenMP ... ... but specific for accelerators
  OpenMP 4.0/4.5: Offloading; compiler support improving (Clang, XL, GCC, ...)

- OpenACC more descriptive, OpenMP more prescriptive
  OpenMP 5.0: Descriptive directive loop

- Same basic principle: Fork/join model
  - Master thread launches parallel child threads
  - Child thread merge after execution
Familiar to OpenMP Programmers

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);
}

More on this later!
Single Code for Multiple Platforms

- Performance portable HPC programming model

OpenPOWER
Sunway
x86 CPU
x86 Xeon Phi
NVIDIA GPU
PEZY-SC

![Graph showing speedup vs single Haswell core for different platforms]

AWE Hydrodynamics CloverLeaf mini-App, bm32 data set

- PGI OpenACC
- Intel OpenMP
- IBM OpenMP

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, RNL: Compiler version: 17.0.1 20161005,
Benchmark: CloverLeaf v1.3 downloaded from http://uk-mac.github.io/CloverLeaf the week of November 7 2016; CloverLeaf_Serial; CloverLeaf_ref (MPI+OpenMP); CloverLeaf_OpenACC (MPI+OpenACC)
Data compiled by PGI November 2016, Volta data collected June 2017
3 Ways to Accelerate Applications

Applications

- Libraries
  - Easy to use
  - Most Performance

- Compiler Directives
  - Easy to use
  - Portable code
  - OpenACC

- Programming Languages
  - Most Performance
  - Most Flexibility
  - CUDA, OpenCL
OpenACC Acceleration Workflow

- Annotate code with directives, indicating parallelism
- OpenACC-capable compiler generates accelerator-specific code
- Success
OpenACC: Key Advantages

- **High-level**
  - Minimal modifications to the code
  - Less than with OpenCL, CUDA, …
  - 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 1.0 and 2.0 and now 2.5 specifications available at http://www.openacc.org
- Quick reference card also available and useful: https://www.openacc.org/resources
- Implementations available now from PGI, Cray, and GCC: https://www.openacc.org/tools
- GCC version of OpenACC in 5.x and 6.x, but use 7.x: https://www.openacc.org/tools
- Best free option is very probably PGI Community Edition: http://www.pgroup.com/products/community.htm
Compilers

- Trust compiler to generate intended parallelism; always check status output!
- No need to know details of accelerator
  - Leave it to expert compiler engineers
- One code can target different accelerators:
  - GPUs, CPUs → Portability

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Targets</th>
<th>Languages</th>
<th>OSS</th>
<th>Free</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA HPC SDK</td>
<td>NVIDIA GPU, CPU</td>
<td>C, C++, Fortran</td>
<td>No</td>
<td>Yes</td>
<td>Best performance</td>
</tr>
<tr>
<td>GCC</td>
<td>NVIDIA GPU, AMD GPU</td>
<td>C, C++, Fortran</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Clang/LLVM</td>
<td>CPU, NVIDIA GPU</td>
<td>C, C++</td>
<td>Yes</td>
<td>Yes</td>
<td>Via Clang</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OpenMP backend Also: flacc</td>
</tr>
</tbody>
</table>
Compiler Flags and Options

- OpenACC compiler support: activate with compile flag
- NVHPC nvc -acc
  -acc=gpu|multicore  Target GPU or CPU
  -acc=gpu -gpu=cc80 Generate Ampere-compatible code
  -gpu=cc80,lineinfo Add source code information in binary
  -gpu=managed Use unified memory
  -Minfo=accel Print acceleration info
- GCC gcc -fopenacc
  -fopenacc-dim=geom Use geom configuration for threads
  -foffload="-lm -O3" Provide flags to offload compiler
  -fopt-info-omp Print acceleration info
Success

- Serial to parallel: fast
- Serial to fast parallel: more time needed
- Start simple → refine
- Expose more and more parallelism
- Because of *generality*:
  - Sometimes last bit of hardware performance difficult
  - But use OpenACC together with other accelerator-targeting techniques (CUDA, libraries, ...)

Lecture 7 – Accelerator Parallel Programming  
CIS 631: Advanced Parallel Computing
OpenACC Accelerator Model

- Main program executes on host
- Device code is transferred to accelerator
- Execution on accelerator is started
- Host waits until return
  - Except for async
- Two separate memory spaces
  - Data transfers back and forth
  - Transfers hidden from programmer
  - Memories not coherent!
- Compiler helps
- GPU runtime helps
Parallelism Workflow

1. Identify available parallelism
2. Parallelize loops with OpenACC
3. Optimize data locality
4. Optimize loop performance
Directive Syntax

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

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

- C

  ```c
  #pragma acc directive [clause [,] clause] ...
  ```

  Often followed by a structured code block
A Simple Example: SAXPY

SAXPY in C

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

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

SAXPY in Fortran

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

... $ Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...
```
Our first OpenACC Directive – kernels

- A kernel is a parallel routine to run on parallel hardware.
- The kernels directive is a request that each loop execute as a separate kernel on the GPU.

```c
#pragma acc kernels
{  
  structured block
}
```

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

- A kernel is parallel routine to run on parallel hardware
- The kernels directive is a request that each loop execute as a separate kernel on the GPU

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

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

!$acc end kernels
```

```fortran
!$acc kernels [clause ...]
  structured block
!$acc end kernels
```
Complete SAXPY Example Code

```c
#include <stdlib.h>

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

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

    float *x = (float*)malloc(N * sizeof(float));
    float *y = (float*)malloc(N * sizeof(float));
    for (int i = 0; i < N; ++i) {
        x[i] = 2.0f;
        y[i] = 1.0f;
    }
    saxpy(N, 3.0f, x, y);

    return 0;
}
```

“I promise y is not aliased by anything else (esp. x)”
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 not parallelize loops that access ptr
  - Note: if programmer violates the declaration, behavior is undefined
Compile and Run

- C:
  
  `pgcc -acc -ta=nvidia -Minfo=accel -o saxpy_acc saxpy.c`

- Fortran:
  
  `pgf90 -acc -ta=nvidia -Minfo=accel -o saxpy_acc saxpy.f90`

- Compiler output:

  `pgcc -acc -Minfo=accel -ta=nvidia -o saxpy_acc saxpy.c`

  `saxpy:
  8, Generating copyin(x[:n-1])
  Generating copy(y[:n-1])
  Generating compute capability 1.0 binary
  Generating compute capability 2.0 binary
  9, Loop is parallelizable
  Accelerator kernel generated
  9, #pragma acc loop worker, vector(256) /* blockIdx.x threadIdx.x */
  CC 1.0 : 4 registers; 52 shared, 4 constant, 0 local memory bytes; 100% occupancy
  CC 2.0 : 8 registers; 4 shared, 64 constant, 0 local memory bytes; 100% occupancy`
OpenACC: Complete SAXPY Example Code

```c
#include <stdlib.h>

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

int main(int argc, char **argv)
{
    int N = 1<<20; // 1 million floats
    if (argc > 1) N = atoi(argv[1]);
    float *x = (float*)malloc(N * sizeof(float));
    float *y = (float*)malloc(N * sizeof(float));
    for (int i = 0; i < N; ++i) {
        x[i] = 2.0f;
        y[i] = 1.0f;
    }
    saxpy(N, 3.0f, x, y);
    return 0;
}
```

CUDA: Partial CUDA C SAXPY Code

```c
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
```

OpenACC versus CUDA Implementations

```c
#include <stdlib.h>

void saxpy(int n, 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) );
cudaMemcpy( (void**)&yd, 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) );
cudaMemcpyyDeviceToHost );
cudaFree( xd ); cudaFree( yd );
}
```
Big Difference!

- CUDA: Hard to Maintain
  - With CUDA, we changed the structure of the old code
  - Non-CUDA programmers cannot 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:
  - If it is this simple, why not just throw kernels in front of every loop?
  - Better yet, why not have 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:
    - data dependencies in loop
    - data movement

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

```c
for(index=0, index<1000000,index++)
    Array[index] = 4 * Array[index];
```
No Data Dependencies

- A run on 1000 processors for the loop below

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

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

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

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

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

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

```c
for(index=5000, index<5999, index++)
    Array[index] = 4 * Array[index];
```

```c
for(index=6000, index<6999, index++)
    Array[index] = 4 * Array[index];
```

```c
for(index=7000, index<7999, index++)
    Array[index] = 4 * Array[index];
```

```c
for(index=8000, index<8999, index++)
    Array[index] = 4 * Array[index];
```

```c
for(index=9000, index<9999, index++)
    Array[index] = 4 * Array[index];
```

```c
for(index=10000, index<100000, 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:

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

- This is a serial code
With Data Dependencies

- 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, …

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

for(index=1000, index<1999,index++)
    Array[1000] = 4 * Array[1000] - Array[999];
```
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.
- 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
  - 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
Laplace Solver (our good old heat equation)

- Laplace equation applies to many physical problems: electrostatics, fluid flow, temperature

\[ \nabla^2 f(x, y) = 0 \]

- For temperature, steady state heat equation
Jacobi Iteration Implementation

- The Laplace equation on a grid states that each grid point is the average of its neighbors

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

- We can iteratively converge to that state by repeatedly computing new values at each point from the average of neighboring points

- We just keep doing this until the difference from one pass to the next is small enough for us to tolerate
Jacobi Iteration C Code

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

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

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

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

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

Iterate until converged
Iterate across matrix elements
Calculate new value from neighbors
Compute max error for convergence
Swap input/output arrays
while (error > tol && iter < iter_max) {
    error=0.0;

    #pragma omp parallel for shared(m, n, Anew, A)
    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]);

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

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

    iter++;
}
OpenMP Fortran Code

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

!$omp parallel do shared(m,n,Anew,A) reduction(max:err)
do j=1,m
   do i=1,n
      Anew(i,j) = .25_fp_kind * (A(i+1, j ) + A(i-1, j ) + &
                                  A(i , j-1) + A(i , j+1))
      err = max(err, Anew(i,j) - A(i,j))
   end do
end do

!$omp parallel do shared(m,n,Anew,A)
do j=1,m-2
   do i=1,n-2
      A(i,j) = Anew(i,j)
   end do
end do

iter = iter +1
end do
**GPU startup overhead**

- If no other GPU process running, GPU driver may be swapped out
  - Linux specific
  - Starting it up can take 1-2 seconds

- Two options
  - Run `nvidia-smi` in persistence mode (requires root permissions)
  - Run “`nvidia-smi -q -l 30`” in the background

- If your running time is off by ~2 seconds from results in these slides, suspect this
  - `Nvidia-smi` should be running in persistent mode for these exercises
First Attempt: OpenACC C

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

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

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

    iter++;
}
```
First Attempt: OpenACC Fortran

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

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

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

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

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

iter = iter +1
end do
```

Generate GPU kernel for loop nest

Generate GPU kernel for loop nest
First Attempt: Compiler output (C)

pgcc  -acc -ta=nvidia -Minfo=accel -o laplace2d_acc laplace2d.c

main:

57, Generating copyin(A[:4095][:4095])
Generating copyout(Anew[1:4094][1:4094])
Generating compute capability 1.3 binary
Generating compute capability 2.0 binary

58, Loop is parallelizable
60, Loop is parallelizable
 Accelerator kernel generated

58, #pragma acc loop worker, vector(16) /* blockIdx.y threadIdx.y */
60, #pragma acc loop worker, vector(16) /* blockIdx.x threadIdx.x */
 Cached references to size [18x18] block of 'A'
 CC 1.3 : 17 registers; 2656 shared, 40 constant, 0 local memory bytes; 75% occupancy
 CC 2.0 : 18 registers; 2600 shared, 80 constant, 0 local memory bytes; 100% occupancy

64, Max reduction generated for error
69, Generating copyout(A[1:4094][1:4094])
Generating copyin(Anew[1:4094][1:4094])
Generating compute capability 1.3 binary
Generating compute capability 2.0 binary

70, Loop is parallelizable
72, Loop is parallelizable
 Accelerator kernel generated

70, #pragma acc loop worker, vector(16) /* blockIdx.y threadIdx.y */
72, #pragma acc loop worker, vector(16) /* blockIdx.x threadIdx.x */
 CC 1.3 : 8 registers; 48 shared, 8 constant, 0 local memory bytes; 100% occupancy
 CC 2.0 : 10 registers; 8 shared, 56 constant, 0 local memory bytes; 100% occupancy
# First Attempt: Performance

**CPU:** Intel Xeon X5680  
6 Cores @ 3.33GHz  

**GPU:** NVIDIA Tesla M2070

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 1 OpenMP thread</td>
<td>69.80</td>
<td>--</td>
</tr>
<tr>
<td>CPU 2 OpenMP threads</td>
<td>44.76</td>
<td>1.56x</td>
</tr>
<tr>
<td>CPU 4 OpenMP threads</td>
<td>39.59</td>
<td>1.76x</td>
</tr>
<tr>
<td>CPU 6 OpenMP threads</td>
<td>39.71</td>
<td>1.76x</td>
</tr>
<tr>
<td>OpenACC GPU</td>
<td>162.16</td>
<td>0.24x FAIL</td>
</tr>
</tbody>
</table>

**Speedup vs. 1 CPU core**  
**Speedup vs. 6 CPU cores**
Basic Concepts

For efficiency, decouple data movement and compute off-load
```c
while ( error > tol && iter < iter_max )
{
  error=0.0;

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

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

**Excessive Data Transfers**

While (error > tol && iter < iter_max)
{
  error = 0.0;

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

- Data needs to be managed

Fortran

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

C

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

General Clauses

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

- **copy ( list )** Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.

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

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

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

- **present ( list )** Data is already present on GPU from another containing data region.

and **present_or_copy[in|out], present_or_create, deviceptr.**
Array Shaping

- Compiler sometimes cannot determine size of arrays
  - Must specify explicitly using data clauses and array “shape”

- C
  ```c
  #pragma acc data copyin(a[0:size-1]),
  copyout(b[s/4:3*s/4])
  ```

- Fortran
  ```fortran
  !$pragma acc data copyin(a(1:size)),
  copyout(b(s/4:3*s/4))
  ```

- Data clauses can be used on data, kernels, or parallel
Update Construct

- Used to update existing data after it has changed in its corresponding copy (e.g., update device copy after host copy changes)
- Move data from GPU to host, or host to GPU
- Data movement can be conditional, and asynchronous

**Fortran**

```
!$acc update [clause ...]
```

**C**

```
#pragma acc update [clause ...]
```

**Clauses**

- `host( list )`
- `device( list )`
- `if( expression )`
- `async( expression )`
Second Attempt: OpenACC C

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

#pragma acc kernels
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++) {
            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] + 
                                A[j-1][i] + A[j+1][i]);
            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }

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

    iter++;
}
```

Copy A in at beginning of loop, out at end. Allocate Anew on accelerator.
### Second Attempt: Performance

**CPU:** Intel Xeon X5680  
6 Cores @ 3.33GHz  

**GPU:** NVIDIA Tesla M2070

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 1 OpenMP thread</td>
<td>69.80</td>
<td>--</td>
</tr>
<tr>
<td>CPU 2 OpenMP threads</td>
<td>44.76</td>
<td>1.56x</td>
</tr>
<tr>
<td>CPU 4 OpenMP threads</td>
<td>39.59</td>
<td>1.76x</td>
</tr>
<tr>
<td>CPU 6 OpenMP threads</td>
<td>39.71</td>
<td>1.76x</td>
</tr>
<tr>
<td>OpenACC GPU</td>
<td>13.65</td>
<td>2.9x</td>
</tr>
</tbody>
</table>

Note: same code runs in 9.78s on NVIDIA Tesla M2090 GPU
Further speedups

- OpenACC gives us more detailed control over parallelization
  - Via gang, worker, and vector clauses

- By understanding more about OpenACC execution model and GPU hardware organization, we can get higher speedups on this code

- By understanding bottlenecks in the code via profiling, we can reorganize the code for higher performance
Finding Parallelism in your code

- (Nested) for loops are best for parallelization
- Large loop counts needed to offset GPU/memcpy overhead
- Iterations of loops must be independent of each other
  - To help compiler: restrict keyword (C), independent clause
- Compiler must be able to figure out sizes of data regions
  - Can use directives to explicitly control sizes
- Pointer arithmetic should be avoided if possible
  - Use subscripted arrays, rather than pointer-indexed arrays.
- Function calls within accelerated region must be inlineable
**Tips and Tricks**

- Profile to find out where time is being spent
- Eliminate pointer arithmetic
- Inline function calls in directives regions
- Use contiguous memory for multi-dimensional arrays
- Use data regions to avoid excessive memory transfers
- Conditional compilation with \_OPENACC macro
CPU and GPU Memory (location matters)

- At the Beginning CPU and GPU memory very distinct, own addresses
- CUDA 4.0
  - Unified Virtual Addressing: pointer from same address pool, but data copy manual
- CUDA 6.0
  - Unified Memory*: Data copy by driver, but whole data at once
- CUDA 8.0
  - Unified Memory (truly): Data copy by driver, page faults on-demand initiate data migrations (Pascal)
- Future
  - Address Translation Service (POWER); Heterogeneous Memory Management (Linux)
Kernels Construct

Fortran

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

C

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

Clauses

```c
if( condition )
async( expression )
```
**Kernels Construct**

Each loop executed as a separate kernel on the GPU

\[
\begin{align*}
!$\text{acc kernels}$
\begin{align*}
do \ i &= 1, n \\
\quad a(i) &= 0.0 \\
\quad b(i) &= 1.0 \\
\quad c(i) &= 2.0 \\
\end{align*}
end do \\
\begin{align*}
do \ i &= 1, n \\
\quad a(i) &= b(i) + c(i) \\
\end{align*}
end do !$\text{acc end kernels}$
\end{align*}
\]
Parallel Construct

- Programmer identifies block containing parallelism
  - Compiler generates offload code

- Program launch creates *gangs* of parallel threads on parallel device

- Implicit barrier at end of parallel region

- Each gang executes same code sequentially
Parallel Construct

Fortran

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

Clauses

```fortran
if( condition )
async( expression )
num_gangs( expression )
num_workers( expression )
vector_length( expression )
```

C

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

```c
private( list )
firstprivate( list )
reduction( operator:list )
```
**Parallel Clauses**

- `num_gangs(expression)`
  - Controls how many parallel gangs are created (CUDA `gridDim`)

- `num_workers(expression)`
  - Controls how many workers are created in each gang (CUDA `blockDim`)

- `vector_length(list)`
  - Controls vector length of each worker (SIMD execution)

- `private(list)`
  - A copy of each variable in list is allocated to each gang

- `firstprivate(list)`
  - `private` variables initialized from host

- `reduction(operator:list)`
  - `private` variables combined across gangs
What’s the difference? kernels versus parallel

- With parallel, you the programmer are defining which loops to parallelize and how
  - Basically you are telling the compiler that it is OK to parallelize particular loops … the compiler does not check
  - Have more explicit control
  - All code within the region is offloaded as one CUDA kernel
  - If you have multiple outer loops within the parallel region, they will still be offloaded in one CUDA kernel

- With kernels, you are defining a region of code which may be parallelized, but it's the compiler's job to then determine which loops to parallelize and how
  - Compiler performs parallel analysis
  - Can cover a large area of code with single directive
  - Multiple loops within this region may be split into a sequence of separate CUDA kernel launches
Loop Construct

- Programmer identifies loop eligible for parallelization
- Directive must be directly before loop
- Describe type of parallelism (optional)

### Fortran

```
!$acc loop [clause …]
  loop
!$acc end loop
```

### C

```
#pragma acc loop [clause …]
  { loop }
```

### Combined directives

```
!$acc parallel loop [clause …]
!$acc kernels loop [clause …]
!$acc parallel loop [clause …]
!$acc kernels loop [clause …]
```
Loop Clauses

collapse( n )

Applies directive to the following n nested loops

seq

Executes the loop sequentially on the GPU

private( list )

A copy of each variable in list is created for each iteration of the loop

reduction( operator:list )

private variables combined across iterations
Loop Clauses Inside parallel Region

- **gang**
  Shares iterations across the gangs of the parallel region

- **worker**
  Shares iterations across the workers of the gang

- **vector**
  Execute the iterations in SIMD mode
Loop Clauses Inside kernels Region

- **gang [ ( num_gangs ) ]**: Shares iterations across across at most `num_gangs` gangs

- **worker [ ( num_workers ) ]**: Shares iterations across at most `num_workers` of a single gang

- **vector [ ( vector_length ) ]**: Execute the iterations in SIMD mode with maximum `vector_length`

- **independent**: Specify that the loop iterations are independent
Example: parallel loop

```c
double sum = 0.0;
#pragma acc parallel loop
for (int i=0; i<N; i++) {
    x[i] = 1.0;
    y[i] = 2.0;
}

#pragma acc parallel loop reduction(+:sum)
for (int i=0; i<N; i++) {
    y[i] = i*x[i]+y[i];
    sum+=y[i];
}
```
### Other Directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache construct</td>
<td>Cache data in software managed data cache (CUDA shared memory)</td>
</tr>
<tr>
<td>host_data construct</td>
<td>Makes the address of device data available on the host</td>
</tr>
<tr>
<td>wait directive</td>
<td>Waits for asynchronous GPU activity to complete</td>
</tr>
<tr>
<td>declare directive</td>
<td>Specify that data is to allocated in device memory for the duration of an implicit data region created during the execution of a subprogram</td>
</tr>
</tbody>
</table>
Runtime Library Routines

Fortran

use openacc
#include "openacc_lib.h"

acc_get_num_devices
acc_set_device_type
acc_get_device_type
acc_set_device_num
acc_get_device_num
acc_async_test
acc_async_test_all

C

#include "openacc.h"

acc_async_wait
acc_async_wait_all
acc_shutdown
acc_on_device
acc_malloc
acc_free
Environment and Conditional Compilation

ACC_DEVICE device

Specifies which device type to connect to.

ACC_DEVICE_NUM num

Specifies which device number to connect to.

_OPENACC

Preprocessor directive for conditional compilation. Set to OpenACC version