Our Workshop Environment

John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center

Our Environment For IHPCSS

Your laptops or workstations: only used for portal access.

Bridges-2 is our HPC platform.

We will here briefly go through the steps to login, edit, compile and run before we get into the real materials.

We want to get all of the distractions and local trivia out of the way here. Everything *after* this part applies to any HPC environment you will encounter.

Getting Connected

- We will be working on bridges2.psc.edu. Use an ssh client (a Putty terminal, for example), to ssh to the machine.

- At this point you are on a login node. It will have a name like “bridges2-login011”. This is a fine place to edit and compile codes. However we must be on compute nodes to do actual computing. We have designed Bridges to be the world’s most interactive supercomputer. We generally only require you to use the batch system when you want to. Otherwise, you get your own personal piece of the machine. To get a single GPU use “interact –gpu”:

```
[urbanic@bridges2-login011]$ interact -gpu
[urbanic@v005]$
```

- However when we have too many of you looking for very quick turnaround, we will fall back on the queuing system to help. We will keep it very simple today:

```
[urbanic@bridges2-login011]$ sbatch gpu.job
```

Editors

For editors, we have several options:

- emacs
- vi
- nano: use this if you aren’t familiar with the others

Compiling

We will be using standard Fortran and C compilers. They should look familiar.

- pgcc for C
- pgf90 for Fortran

Note that on Bridges you would normally have to enable this compiler with

```
module load pgi/nvhpc
```

I have put that in the .bashrc file that we will all start with.

Multiple Sessions

There is no reason not to open other sessions (windows) to the login nodes for compiling and editing. You may find this convenient. Feel free to do so.

Our Setup For This Workshop

After you copy the files from the training directory, you will have:

```
/Exercises
        /Test
        /OpenMP
                laplace_serial.f90/c
                /Solutions
                /Examples
                /Prime
        /OpenACC
        /MPI
```

Preliminary Exercise

Let's get the boring stuff out of the way now.

- **Log on to Bridges.**

  ```
  ssh username@bridges2.psc.edu
  ```

- **Run the setup script that will copy over the Exercises directory we will all use. It will also automatically load the right compiler using your .bashrc script whenever you login.**

  ```
  ~training/Setup
  ```

- **As told, logout and log back on again to complete the setup. You won't need to do that in the future.**

- **Edit a file to make sure you can do so. Use emacs, vi or nano (if the first two don't sound familiar).**

- **cd into your exercises/test directory and compile (C or Fortran)**

  ```
  cd Exercises/Test
  nvc test.c
  nvfortran test.f90
  ```

- **Run your program (or just *interact -gpu* and then run *a.out*, if we can all fit)**

  ```
  sbatch gpu.job
  ```

  (Wait a minute, or see how your job is doing with *squeue –u username*)

- **Look at the results**

  ```
  more slurm-55838.out        (The exact job number will differ)
  ```

  It should say *"Congratulations!"*

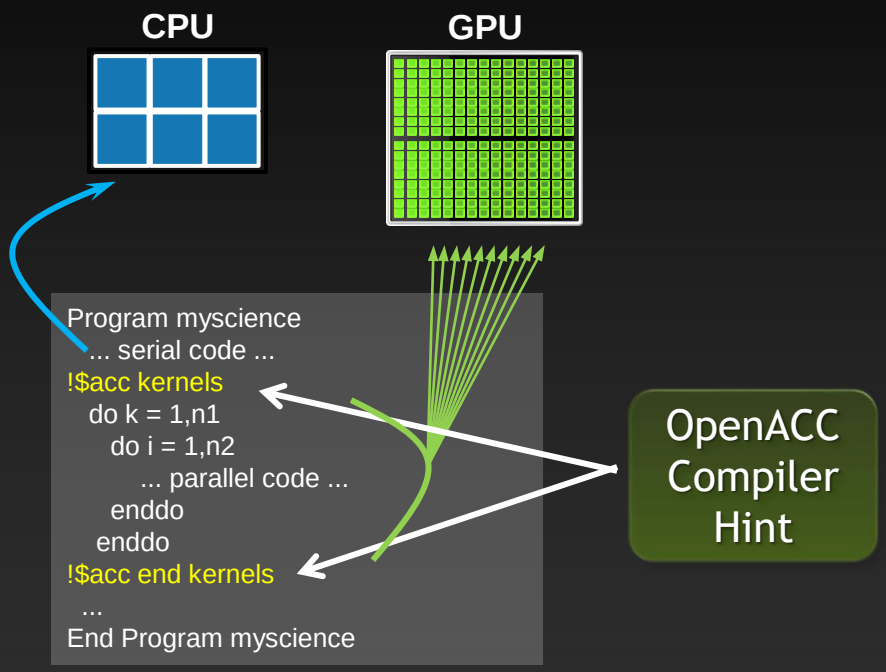
Introduction to OpenACC

John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center

What is OpenACC?

It is a directive based standard to allow developers to take advantage of accelerators such as GPUs from NVIDIA and AMD, Intel's Xeon Phi, FPGAs, and even DSP chips.

Directives



Your original Fortran or C code

Simple compiler hints from coder.

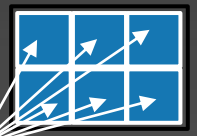
Compiler generates parallel threaded code.

Ignorant compiler just sees some comments.

Familiar to OpenMP Programmers

OpenMP

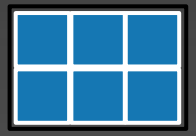
CPU



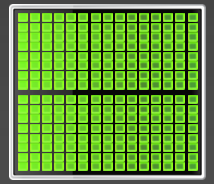
```
main() {
  double pi = 0.0; long i;


  #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);
}
```

OpenACC

CPU



GPU



```
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);
}
```

More on this later!

How Else Would We Accelerate Applications?

Applications

Libraries	OpenACC Directives	Programming Languages (CUDA)
"Drop-in" Acceleration	Incrementally Accelerate Applications	Maximum Flexibility

Key Advantages Of This Approach

- High-level. No involvement of OpenCL, CUDA, etc.
- Single source. No forking off a separate GPU code. Compile the same program for accelerators or serial; non-GPU programmers can play along.
- Efficient. Experience shows very favorable comparison to low-level implementations of same algorithms.
- Performance portable. Supports 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 Standard

- Full OpenACC specifications (now on 3.0) available online

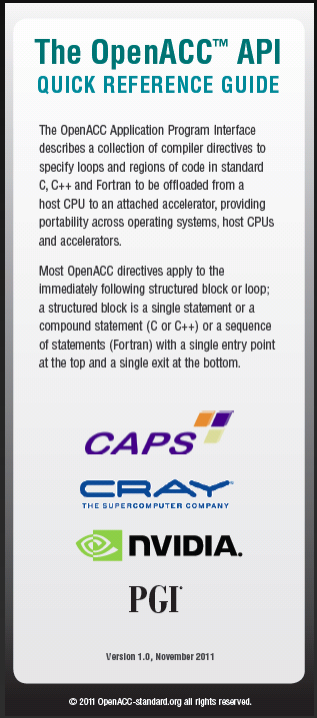
  http://www.openacc-standard.org

- Quick reference card also available and useful

- Implementations available now from PGI, Cray, CAPS and GCC.

- GCC version of OpenACC started in 5.x, but use 10.x

- Best free option is very probably PGI Community version:
  http://www.pgroup.com/products/community.htm



The OpenACC™ API
QUICK REFERENCE GUIDE

The OpenACC Application Program Interface describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators.

Most OpenACC directives apply to the immediately following structured block or loop; a structured block is a single statement or a compound statement (C or C++) or a sequence of statements (Fortran) with a single entry point at the top and a single exit at the bottom.

CAPS
CRAY THE SUPERCOMPUTER COMPANY
NVIDIA.
PGI

Version 1.0, November 2011

© 2011 OpenACC-standard.org all rights reserved.

OPENACC Resources

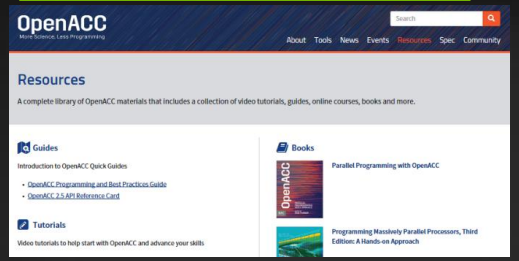
Guides ● Talks ● Tutorials ● Videos ● Books ● Spec ● Code Samples ● Teaching Materials ● Events ● Success Stories ● Courses ● Slack ● Stack Overflow

FREE Compilers

PGI Community EDITION

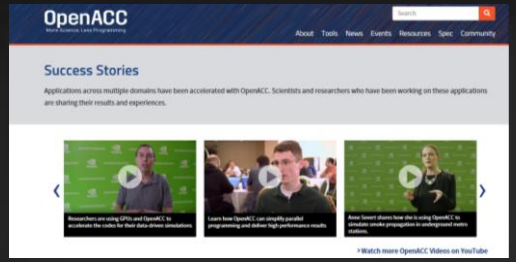
Resources

https://www.openacc.org/resources



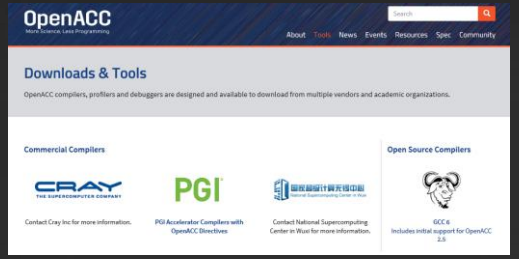
Success Stories

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



Compilers and Tools

https://www.openacc.org/tools



Events

https://www.openacc.org/events

PITTSBURGH SUPERCOMPUTING CENTER

Serious Adoption

NEW PLATFORMS

OpenPOWER™

Sunway TaihuLight
Built around
OpenACC

GROWING COMMUNITY

- 6,000+ enabled developers
- Hackathons constantly
- Diverse online community

PORTING SUCCESS

- Five of 13 CAAR codes using OpenACC
- Gaussian ported to Tesla with OpenACC
- FLUENT using OpenACC in R18 production release

A Few Cases

Reading DNA nucleotide sequences

Shanghai JiaoTong University

4 directives

16x faster

Designing circuits for quantum computing

UIST, Macedonia

1 week

40x faster

Extracting image features in real-time

Aselsan

3 directives

4.1x faster

HydroC- Galaxy Formation

PRACE Benchmark Code, CAPS

1 week

3x faster

Real-time Derivative Valuation

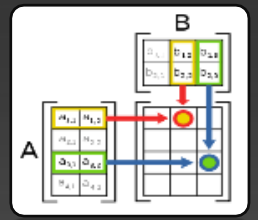
Opel Blue, Ltd

Few hours

70x faster

Matrix Matrix Multiply

Independent Research Scientist



4 directives

6.4x faster

A Champion Case

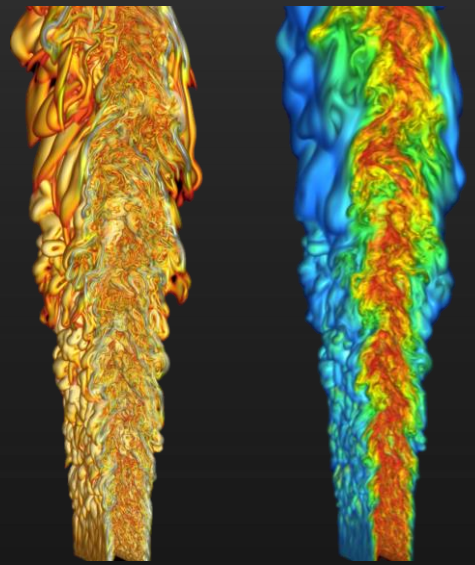
4x Faster

Jaguar	Titan
42 days	10 days

Modified <1%
Lines of Code

15 PF! One of fastest simulations ever!

Design alternative fuels with up to 50% higher efficiency



S3D: Fuel Combustion

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];
}

...
// Somewhere in main
// call 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


...
$ 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 GPU

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.

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)"

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

Compile and Run

- C: `nvc -acc -Minfo=accel saxpy.c`
- Fortran: `nvfortran -acc -Minfo=accel saxpy.f90`

Compiler Output

```
nvc -acc -Minfo=accel 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
         Generating Tesla code
          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
```

- Run: `a.out`

Compare: Partial CUDA C SAXPY Code

Just the subroutine

```
__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 );
}
```

Compare: Partial CUDA Fortran SAXPY Code

Just the subroutine

```
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
```

Again: Complete SAXPY Example Code

Main 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;
}
```

Entire Subroutine

```
#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];
}
```

Big Difference!

- 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.
- We have separate sections for the host code and the GPU code. Different flow of code. Serial path now gone forever.
- 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.
- 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…

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

The answer is that there are two general issues that prevent the compiler from being able to just automatically parallelize every loop.

- Data Dependencies in Loops
- 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:

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

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

No Data Dependency

Processor 0

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

Processor 1

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

Processor 2

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

Processor 3

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

Processor 4

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

Data Dependency

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];
```

This is perfectly valid serial code.

Data Dependency

Now Processor 1, in trying to calculate its first iteration…

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

needs the result of Processor 0's last iteration. If we want the correct ("same as serial") result, we need to wait until processor 0 finishes. Likewise for processors 2, 3, …

Data Dependencies

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

```
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?

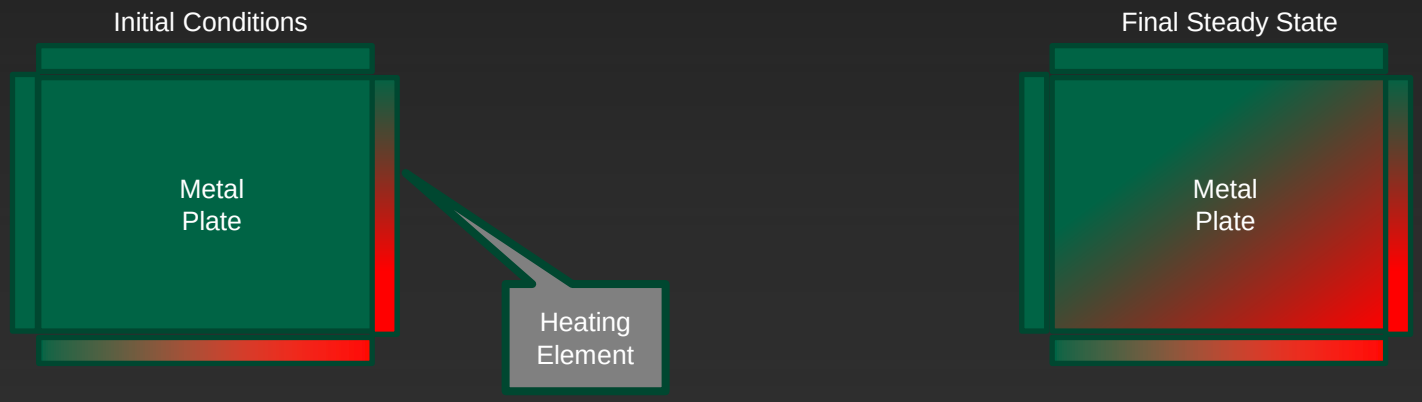
Data Dependencies

- 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.

Our Foundation Exercise: Laplace Solver

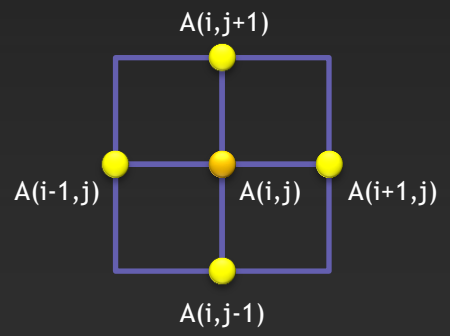
- I've been using this for MPI, OpenMP and now OpenACC. It is a great simulation problem, not rigged for OpenACC.
- In this most basic form, it solves the Laplace equation: $\nabla^2 f(x, y) = 0$
- The Laplace Equation applies to many physical problems, including:
  - Electrostatics
  - Fluid Flow
 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

- For temperature, it is the Steady State Heat Equation:



Exercise Foundation: Jacobi Iteration

- The Laplace equation on a grid states that each grid point is the average of it's neighbors.
- 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.



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

Serial Code Implementation

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

```
do j=1,columns
     do i=1,rows
          temperature(i,j)= 0.25 * (temperature_last(i+1,j)+temperature_last(i-1,j) + &
                                    temperature_last(i,j+1)+temperature_last(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++) {
               Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                           Temperature_last[i][j+1] + Temperature_last[i][j-1]);
          }
     }

     dt = 0.0;

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

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

     iteration++;

}
```

Done?

Calculate

Update temp array and find max change

Output

Serial C Code Subroutines

```c
void initialize(){

    int i,j;

    for(i = 0; i <= ROWS+1; i++){
        for (j = 0; j <= COLUMNS+1; j++){
            Temperature_last[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++) {
        Temperature_last[i][0] = 0.0;
        Temperature_last[i][COLUMNS+1] = (100.0/ROWS)*i;
    }

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

```c
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,Temperature[i][i]);
  }
  printf("\n");
}
```

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

Whole C Code

```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++) {
                Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                            Temperature_last[i][j+1] + Temperature_last[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(Temperature[i][j]-Temperature_last[i][j]), dt);
              Temperature_last[i][j] = Temperature[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++){
            Temperature_last[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++) {
        Temperature_last[i][0] = 0.0;
        Temperature_last[i][COLUMNS+1] = (100.0/ROWS)*i;
    }

    // set top to 0 and bottom to linear increase
    for(j = 0; j <= COLUMNS+1; j++) {
        Temperature_last[0][j] = 0.0;
        Temperature_last[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, Temperature[i][i]);
    }
    printf("\n");
}
```

Serial Fortran Code (kernel)

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

  do j=1,columns
     do i=1,rows
        temperature(i,j)=0.25*(temperature_last(i+1,j)+temperature_last(i-1,j)+ &
                               temperature_last(i,j+1)+temperature_last(i,j-1) )
     enddo
  enddo

  dt=0.0

  do j=1,columns
     do i=1,rows
        dt = max( abs(temperature(i,j) - temperature_last(i,j)), dt )
        temperature_last(i,j) = temperature(i,j)
     enddo
  enddo

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

  iteration = iteration+1

enddo
```

Done?

Calculate

Update temp array and find max change

Output

Serial Fortran Code Subroutines

```
subroutine initialize( temperature_last )
      implicit none

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

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

      temperature_last = 0.0

      !these boundary conditions never change throughout run

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

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

end subroutine initialize
```

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

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

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

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

Whole Fortran Code

```fortran
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) :: temperature, temperature_last

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

      call cpu_time(start_time)      !Fortran timer

      call initialize(temperature_last)

      !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
               temperature(i,j)=0.25*(temperature_last(i+1,j)+temperature_last(i-1,j)+ &
                                      temperature_last(i,j+1)+temperature_last(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(temperature(i,j) - temperature_last(i,j)), dt )
               temperature_last(i,j) = temperature(i,j)
            enddo
         enddo

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

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

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

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

      temperature_last = 0.0

      !these boundary conditions never change throughout run

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

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

end subroutine initialize

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

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

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

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

PITTSBURGH SUPERCOMPUTING CENTER

Exercises: General Instructions for Compiling

- Exercises are in the “Exercises/OpenACC” directory in your home directory
- Solutions are in the “Solutions” subdirectory
- To compile

```
nvc -acc laplace.c
nvfortran -acc laplace.f90
```

- This will generate the executable a.out

Exercises: Very useful compiler option

Adding -Minfo=accel to your compile command will give you some very useful information about how well the compiler was able to honor your OpenACC directives.

```
[urbanic@gpu017 Solutions]$ nvc -acc -Minfo=accel laplace_acc.c
main:
     59, Generating create(Temperature[:][:]) [if not already present]
         Generating copy(Temperature_last[:][:]) [if not already present]
     64, Loop is parallelizable
     65, Loop is parallelizable
         Generating Tesla code
         64, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         65, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
     75, Loop is parallelizable
     76, Loop is parallelizable
         Generating Tesla code
         75, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         76, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
         77, Generating implicit reduction(max:dt)
     85, Generating update self(Temperature[:][:])
```

Special Instructions for Running on the GPUs (during this workshop)

As mentioned, on Bridges2 you generally only have to use the queueing system when you want to. However, as we have hundreds of you wanting quick turnaround, we will have to use it today.

Once you have an a.out that you want to run, you should use the simple job that we have already created (in Exercises/OpenACC) for you to run:

```
fred@bridges2-login011$ sbatch gpu.job
```

Output From Your Batch Job

The machine will tell you it submitted a batch job, and you can await your output, while will come back in a file with the corresponding number as a name:

slurm-138555.out

As everything we are doing this afternoon only requires a few minutes at most (and usually just seconds), you could just sit there and wait for the file to magically appear. At which point you can "more" it or review it with your editor.

Changing Things Up

If you get impatient, or want to see what the machine us up to, you can look at the situation with squeue.

You might wonder what happened to the interaction count that the user is prompted for. I stuck a reasonable default (4000 iterations) into the job file. You can edit it if you want to. The whole job file is just a few lines.

Congratulations, you are now a Batch System veteran. Welcome to supercomputing.

Exercise 1: Using `kernels` to parallelize the main loops

(About 20 minutes)

Q: Can you get a speedup with just the kernels directives?

1. Edit *laplace_serial.c*/f90
    1. Maybe copy your intended OpenACC version to *laplace_acc.c* to start
    2. Add directives where it helps

2. Compile with OpenACC parallelization
    1. nvc -acc -Minfo=accel laplace_acc.c or
       nvfortran -acc -Minfo=accel laplace_acc.f90
    2. Look at your compiler output to make sure you are having an effect

3. Run
    1. sbatch gpu.job (Leave it at 4000 iterations if you want a solution that converges to current tolerance)
    2. Look at output in file that returns (something like slurm-138555.out)
    3. Compare the serial and your OpenACC version for performance difference

Exercise 1 C Solution

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

    #pragma acc kernels
    for(i = 1; i <= ROWS; i++) {
        for(j = 1; j <= COLUMNS; j++) {
            Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                        Temperature_last[i][j+1] + Temperature_last[i][j-1]);
        }
    }

    dt = 0.0; // reset largest temperature change

    #pragma acc kernels
    for(i = 1; i <= ROWS; i++){
        for(j = 1; j <= COLUMNS; j++){
            dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
            Temperature_last[i][j] = Temperature[i][j];
        }
    }

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

    iteration++;
}
```

Generate a GPU kernel

Generate a GPU kernel

Exercise 1 Fortran Solution

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

    !$acc kernels
    do j=1,columns
       do i=1,rows
          temperature(i,j)=0.25*(temperature_last(i+1,j)+temperature_last(i-1,j)+ &
                                 temperature_last(i,j+1)+temperature_last(i,j-1) )
       enddo
    enddo
    !$acc end kernels

    dt=0.0

    !$acc kernels
    do j=1,columns
       do i=1,rows
          dt = max( abs(temperature(i,j) - temperature_last(i,j)), dt )
          temperature_last(i,j) = temperature(i,j)
       enddo
    enddo
    !$acc end kernels

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

    iteration = iteration+1

enddo
```

Generate a GPU kernel

Generate a GPU kernel

Exercise 1: Compiler output (C)

```
[urbanic@gpu017 Solutions]$ nvc -acc -Minfo=accel laplace_acc.c
main:
     59, Generating create(Temperature[:][:]) [if not already present]
         Generating copy(Temperature_last[:][:]) [if not already present]
     64, Loop is parallelizable
     65, Loop is parallelizable
         Generating Tesla code
         64, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         65, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
     75, Loop is parallelizable
     76, Loop is parallelizable
         Generating Tesla code
         75, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         76, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
         77, Generating implicit reduction(max:dt)
     85, Generating update self(Temperature[:][:])
```

Compiler was able to parallelize

Compiler was able to parallelize

First, about that “reduction”

```
while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {
     #pragma acc kernels
     for(i = 1; i <= ROWS; i++) {
          for(j = 1; j <= COLUMNS; j++) {
               Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                           Temperature_last[i][j+1] + Temperature_last[i][j-1]);
          }
     }

     dt = 0.0;

     #pragma acc loop reduction (max:dt)
     for(i = 1; i <= ROWS; i++){
          for(j = 1; j <= COLUMNS; j++){
               dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
               Temperature_last[i][j] = Temperature[i][j];
          }
     }

     .
     .
     iteration++;
}
```

This explicitly declares the reduction.

Exiting this loop, each processor has a different idea of what the max dt is.

With *kernel* the compiler recognizes this and does a reduction, a very convenient thing. We can get too sophisticated for this *autoscoping* to happen.

Exercise 1: Performance

3372 steps to convergence

Execution	Time (s)	Speedup
CPU Serial	20.6	--
CPU 2 OpenMP threads	10.3	2.0
CPU 4 OpenMP threads	5.2	4.0
CPU 8 OpenMP threads	2.6	7.9
CPU 16 OpenMP threads	1.4	14.7
CPU 32 OpenMP threads	0.80	25.7
CPU 64 OpenMP threads	0.72	28.6
CPU 128 OpenMP threads	1.4	14.7
OpenACC GPU	32.4	0.6x

Using NVHPC 21.2 on a V100

What's with the OpenMP?

- We can compare our GPU results to the best the multi-core CPUs can do.

- If you are familiar with OpenMP, or even if you are not, you can compile and run the OpenMP enabled versions in your OpenMP directory as:

```
nvc -mp laplace_omp.c     or    nvfortran -mp laplace_omp.f90
```

then to run on 8 threads do:

```
export OMP_NUM_THREADS=8
a.out
```

- Note that you probably only have 8 real cores if you are still on a GPU node. Do something like "interact -n28" if you want a full node of cores.

What went wrong?

`export PGI_ACC_TIME=1` to activate profiling and run again:

```
Accelerator Kernel Timing data
/home/urbanic/laplace_bad_acc.c
  main  NVIDIA  devicenum=0
    time(us): 12,095,531
    62: compute region reached 3372 times
        64: kernel launched 3372 times
            grid: [32x250]  block: [32x4]
             device time(us): total=127,989 max=48 min=37 avg=37
            elapsed time(us): total=241,221 max=1,407 min=61 avg=71
    62: data region reached 6744 times
        62: data copyin transfers: 3372
             device time(us): total=2,446,765 max=972 min=712 avg=725
        70: data copyout transfers: 3372
             device time(us): total=2,098,635 max=835 min=616 avg=622
    73: compute region reached 3372 times
        73: data copyin transfers: 3372
             device time(us): total=32,465 max=71 min=6 avg=9
        75: kernel launched 3372 times
            grid: [32x250]  block: [32x4]
             device time(us): total=179,342 max=63 min=52 avg=53
            elapsed time(us): total=294,686 max=407 min=76 avg=87
        75: reduction kernel launched 3372 times
            grid: [1]  block: [256]
             device time(us): total=50,490 max=23 min=14 avg=14
            elapsed time(us): total=137,910 max=549 min=34 avg=40
        75: data copyout transfers: 3372
             device time(us): total=60,080 max=266 min=13 avg=17
    73: data region reached 6744 times
        73: data copyin transfers: 6744
             device time(us): total=5,004,411 max=1,005 min=716 avg=742
        82: data copyout transfers: 3372
             device time(us): total=2,095,354 max=854 min=616 avg=621
```

0.2 seconds

0.3 seconds

0.1 seconds

2.4 seconds

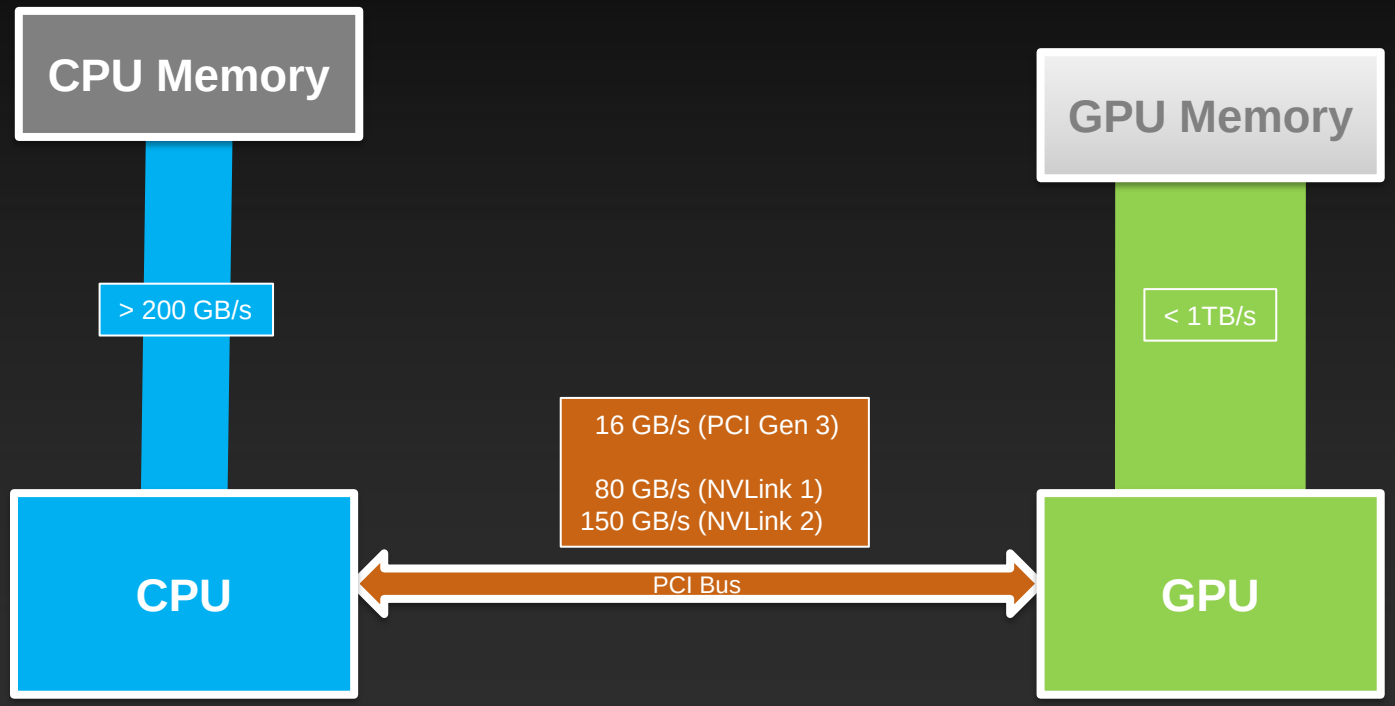
2.0 seconds

5.0 seconds

2.0 seconds

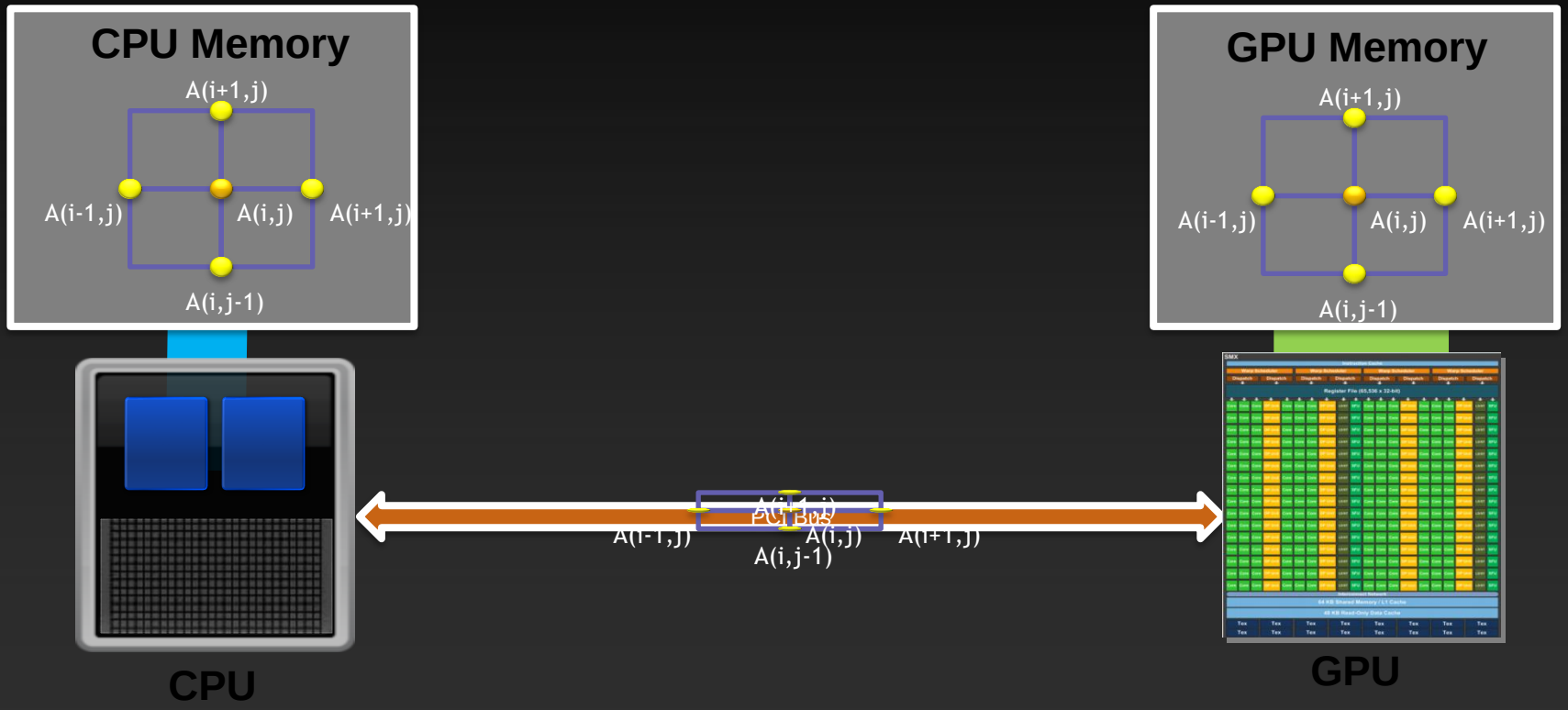
Basic Concept

Simplified, but sadly true



All bandwidths one-direction.

Multiple Times Each Iteration



Excessive Data Transfers

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

Temperature, Temperature_old resident on host

Temperature, Temperature_old resident on device

```
#pragma acc kernels
for(i = 1; i <= ROWS; i++) {
   for(j = 1; j <= COLUMNS; j++) {
      Temperature[i][j] = 0.25 * (Temperature_old[i+1][j] + …
   }
}
```

Temperature, Temperature_old resident on device

Temperature, Temperature_old resident on host

4 copies happen every iteration of the outer while loop!

```
dt = 0.0;
```

Temperature, Temperature_old resident on host

Temperature, Temperature_old resident on device

```
#pragma acc kernels
for(i = 1; i <= ROWS; i++) {
   for(j = 1; j <= COLUMNS; j++) {
      Temperature[i][j] = 0.25 * (Temperature_old[i+1][j] + …
```

Temperature, Temperature_old resident on device

Temperature, Temperature_old resident on host

```
}
```

Data Management

The First, Most Important, and Possibly Only OpenACC Optimization

Scoped Data Construct Syntax

Fortran

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

C

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

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

- Compilers sometimes cannot determine the size of arrays, so we must specify explicitly using data clauses with an array "shape". The compiler will let you know if you need to do this. Sometimes, you will want to for your own efficiency reasons.

- C

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

- Fortran

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

- Fortran uses start:end and C uses start:length
- Data clauses can be used on `data`, `kernels` or `parallel`

Compiler will (increasingly) often make a good guess…

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

  int i;
  double A[2000], B[1000], C[1000];
    .
    .
    .
  #pragma acc kernels
  for (i=0; i<1000; i++){

    A[i] = 4 * i;
    B[i] = B[i] + 2;
    C[i] = A[i] + 2 * B[i];

  }
    .
    .
    .
}
```

Smarter

Smartest

```
nvc -acc -Minfo=accel loops.c
main:
   6, Generating present_or_copyout(C[:])
      Generating present_or_copy(B[:])
      Generating present_or_copyout(A[:1000])
      Generating NVIDIA code
   7, Loop is parallelizable
      Accelerator kernel generated
```

Data Regions Have Real Consequences

Simplest Kernel

```
int main(int argc, char** argv){

float A[1000];

    #pragma acc kernels
    for( int iter = 1; iter < 1000 ; iter++){
      A[iter] = 1.0;
    }

    A[10] = 2.0;

  printf("A[10] = %f", A[10]);
}
```

A[] Copied To GPU

A[] Copied To Host

Runs On Host

Output:

A[10] = 2.0

With Global Data Region

```
int main(int argc, char** argv){

float A[1000];

 #pragma acc data copy(A)
  {

    #pragma acc kernels
    for( int iter = 1; iter < 1000 ; iter++){
      A[iter] = 1.0;
    }

    A[10] = 2.0;

  }

  printf("A[10] = %f", A[10]);
}
```

A[] Copied To GPU

Still Runs On Host

A[] Copied To Host

Output:

A[10] = 1.0

Data Regions Are Different Than Compute Regions

Compute Region

```
int main(int argc, char** argv){

float A[1000];

#pragma acc data copy(A)
 {

   #pragma acc kernels
   for( int iter = 1; iter < 1000 ; iter++){
     A[iter] = 1.0;
   }

   A[10] = 2.0;

 }

 printf("A[10] = %f", A[10]);
}
```

Data Region

Output:

A[10] = 1.0

Data Movement Decisions

- Much like loop data dependencies, sometime the compiler needs your human intelligence to make high-level decisions about data movement. Otherwise, it must remain conservative - sometimes at great cost.

- You must think about when data truly needs to migrate, and see if that is better than the default.

- Besides the scope-based data clauses, there are OpenACC options to let us manage data movement more intensely or asynchronously. We could manage the above behavior with the `update` construct:

Fortran :

```
!$acc update [host(), device(), …]
```

C:

```
#pragma acc update [host(), device(), …]
```

Ex: `#pragma acc update host(Temp_array)  //Get host a copy from device`

Exercise 2: Use acc `data` to minimize transfers

(about 40 minutes)

Q: What speedup can you get with data + kernels directives?

- Start with your Exercise 1 solution or grab laplace_bad_acc.c/f90 from the Solutions subdirectory. This is just the solution of the last exercise.
- Add *data* directives where it helps.
  - Think: when *should* I move data between host and GPU? Think how you would do it by hand, then determine which data clauses will implement that plan.
  - Hint: you may find it helpful to ignore the output at first and just concentrate on getting the solution to converge quickly (at 3372 steps). Then worry about *updating* the printout.

Exercise 2 C Solution

```
#pragma acc data copy(Temperature_last, Temperature)
while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

    // main calculation: average my four neighbors
    #pragma acc kernels
    for(i = 1; i <= ROWS; i++) {
        for(j = 1; j <= COLUMNS; j++) {
            Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                        Temperature_last[i][j+1] + Temperature_last[i][j-1]);
        }
    }

    dt = 0.0; // reset largest temperature change

    // copy grid to old grid for next iteration and find latest dt
    #pragma acc kernels
    for(i = 1; i <= ROWS; i++){
        for(j = 1; j <= COLUMNS; j++){
          dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
          Temperature_last[i][j] = Temperature[i][j];
        }
    }

    // periodically print test values
    if((iteration % 100) == 0) {
        #pragma acc update host(Temperature)
        track_progress(iteration);
    }

    iteration++;
}
```

No data movement in this block.

Except once in a while here.

Exercise 2, Slightly better solution

```
#pragma acc data copy(Temperature_last), create(Temperature)
while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

    // main calculation: average my four neighbors
    #pragma acc kernels
    for(i = 1; i <= ROWS; i++) {
        for(j = 1; j <= COLUMNS; j++) {
            Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                        Temperature_last[i][j+1] + Temperature_last[i][j-1]);
        }
    }

    dt = 0.0; // reset largest temperature change

    // copy grid to old grid for next iteration and find latest dt
    #pragma acc kernels
    for(i = 1; i <= ROWS; i++){
        for(j = 1; j <= COLUMNS; j++){
          dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
          Temperature_last[i][j] = Temperature[i][j];
        }
    }

    // periodically print test values
    if((iteration % 100) == 0) {
        #pragma acc update host(Temperature)
        track_progress(iteration);
    }

    iteration++;
}
```

Temperature is purely temporary.

Slightly better still solution

```
#pragma acc data copy(Temperature_last), create(Temperature)
while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

    // main calculation: average my four neighbors
    #pragma acc kernels
    for(i = 1; i <= ROWS; i++) {
        for(j = 1; j <= COLUMNS; j++) {
            Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                        Temperature_last[i][j+1] + Temperature_last[i][j-1]);
        }
    }

    dt = 0.0; // reset largest temperature change

    // copy grid to old grid for next iteration and find latest dt
    #pragma acc kernels
    for(i = 1; i <= ROWS; i++){
        for(j = 1; j <= COLUMNS; j++){
          dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
          Temperature_last[i][j] = Temperature[i][j];
        }
    }

    // periodically print test values
    if((iteration % 100) == 0) {
        #pragma acc update host(Temperature[ROWS-4:5][COLUMNS-4:5])
        track_progress(iteration);
    }

    iteration++;
}
```

Only need corner elements.

Exercise 2 Fortran Solution

```
!$acc data copy(temperature_last), create(temperature)
do while ( dt > max_temp_error .and. iteration <= max_iterations)

   !$acc kernels
   do j=1,columns
      do i=1,rows
         temperature(i,j)=0.25*(temperature_last(i+1,j)+temperature_last(i-1,j)+ &
                                temperature_last(i,j+1)+temperature_last(i,j-1) )
      enddo
   enddo
   !$acc end kernels

   dt=0.0

   !copy grid to old grid for next iteration and find max change
   !$acc kernels
   do j=1,columns
      do i=1,rows
         dt = max( abs(temperature(i,j) - temperature_last(i,j)), dt )
         temperature_last(i,j) = temperature(i,j)
      enddo
   enddo
   !$acc end kernels

   !periodically print test values
   if( mod(iteration,100).eq.0 ) then
      !$acc update host(temperature)
      call track_progress(temperature, iteration)
   endif

   iteration = iteration+1

enddo
!$acc end data
```

Keep these on GPU

Extra efficient:

!$acc update host(temperature(columns-5:columns,rows-5:rows))

Except bring back a copy here

Exercise 1: Performance

3372 steps to convergence

Execution	Time (s)	Speedup
CPU Serial	20.6	--
CPU 2 OpenMP threads	10.3	2.0
CPU 4 OpenMP threads	5.2	4.0
CPU 8 OpenMP threads	2.6	7.9
CPU 16 OpenMP threads	1.4	14.7
CPU 32 OpenMP threads	0.80	25.7
CPU 64 OpenMP threads	0.72	28.6
CPU 128 OpenMP threads	1.4	14.7
OpenACC GPU	32.4	0.6x

Using NVHPC 21.2 on a V100

OpenACC or OpenMP?

Don't draw any grand conclusions yet. We have gotten impressive speedups from both approaches. But our problem size is pretty small. Our main data structure is:

1000 x 1000 = 1M elements = 8MB of memory

We have 2 of these (temperature and temperature_last) so we are using roughly 16 MB of memory. Not very large. When divided over cores it gets even smaller and can easily fit into cache.

The algorithm is realistic, but the problem size is tiny and hence the memory bandwidth stress is very low.

OpenACC or OpenMP on Larger Data?

We can easily scale this problem up, so why don't I? Because it is nice to have exercises that finish in a few minutes or less.

We scale this up to 10K x 10K (1.6 GB problem size) for the hybrid challenge. These numbers start to look a little more realistic. But the serial code takes over 30 minutes to finish. That would have gotten us off to a slow start!

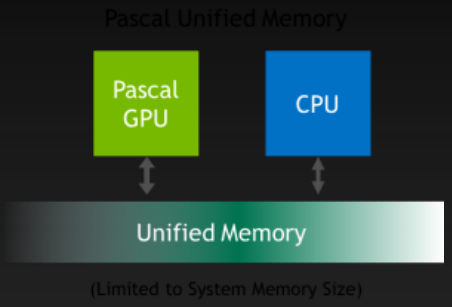
Execution	Time (s)	Speedup
CPU Serial	2187	--
CPU 16 OpenMP threads	183	12
CPU 28 OpenMP threads	162	13.5
OpenACC	103	21

Obvious cusp for core scaling appears

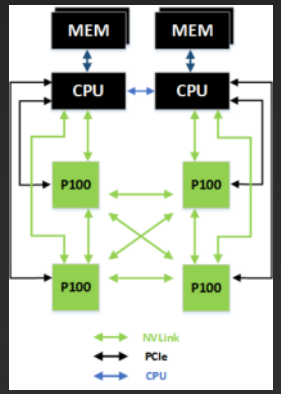
10K x 10K Problem Size

Latest Happenings In Data Management

- **Unified Memory**
  - Unified address space allows us to pretend we have shared memory
  - Skip data management, hope it works, and then optimize if necessary
  - For dynamically allocated memory can eliminate need for pointer clauses



- **NVLink**
  - One route around PCI bus (with multiple GPUs)



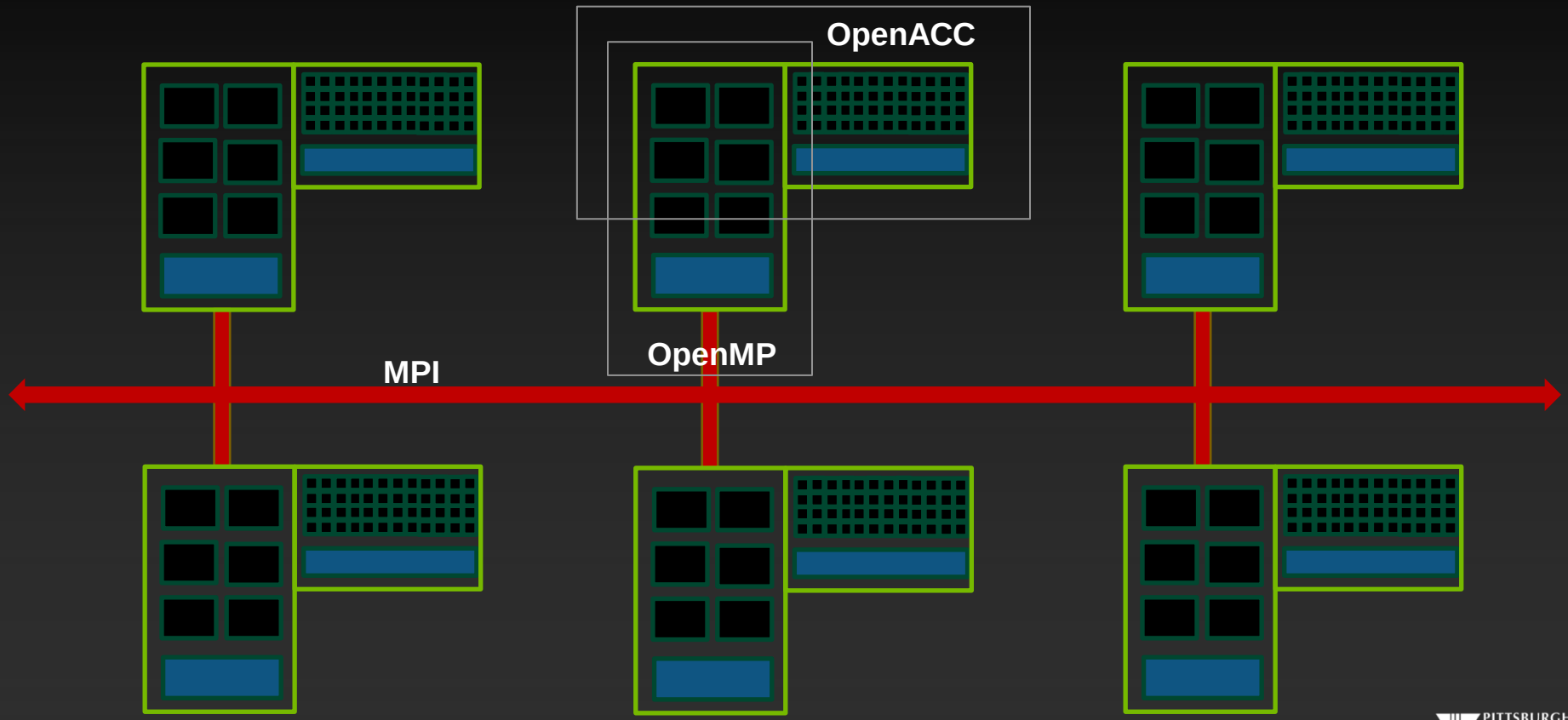
Further speedups

- OpenACC gives us even 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
- But you have already gained most of any potential speedup, and you did it with a few lines of directives!

Is OpenACC Living Up To My Claims?

- High-level. No involvement of OpenCL, CUDA, etc.
- Single source. No forking off a separate GPU code. Compile the same program for accelerators or serial; non-GPU programmers can play along.
- Efficient. Experience shows very favorable comparison to low-level implementations of same algorithms. kernels is magical!
- Performance portable. Supports 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.*

In Conclusion…



Advanced OpenACC

John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center

Outline

Loop Directives

Data Declaration Directives

Data Regions Directives

Cache directives

Wait / update directives

Runtime Library Routines

Environment variables

.

.

.

Targeting the Architecture (But Not Admitting It)

Part of the awesomeness of OpenACC has been that you have been able to ignore the hardware specifics. But, now that you know a little bit more about CUDA/GPU architecture, you might suspect that you can give the compiler still more help in optimizing. In particular, you might know the hardware specifics of a particular model. The compiler might only know which "family" it is compiling for (Fermi, Kepler, Pascal etc.).

Indeed, the OpenACC spec has methods to target architecture hierarchies, and not just GPUs (think Intel MIC). Let's see how they map to what we know about GPUs.

V100 GPU and SM

Volta GV100 GPU with 85 Streaming Multiprocessor (SM) units

Volta GV100 SM

Turing Memory Hierarchy

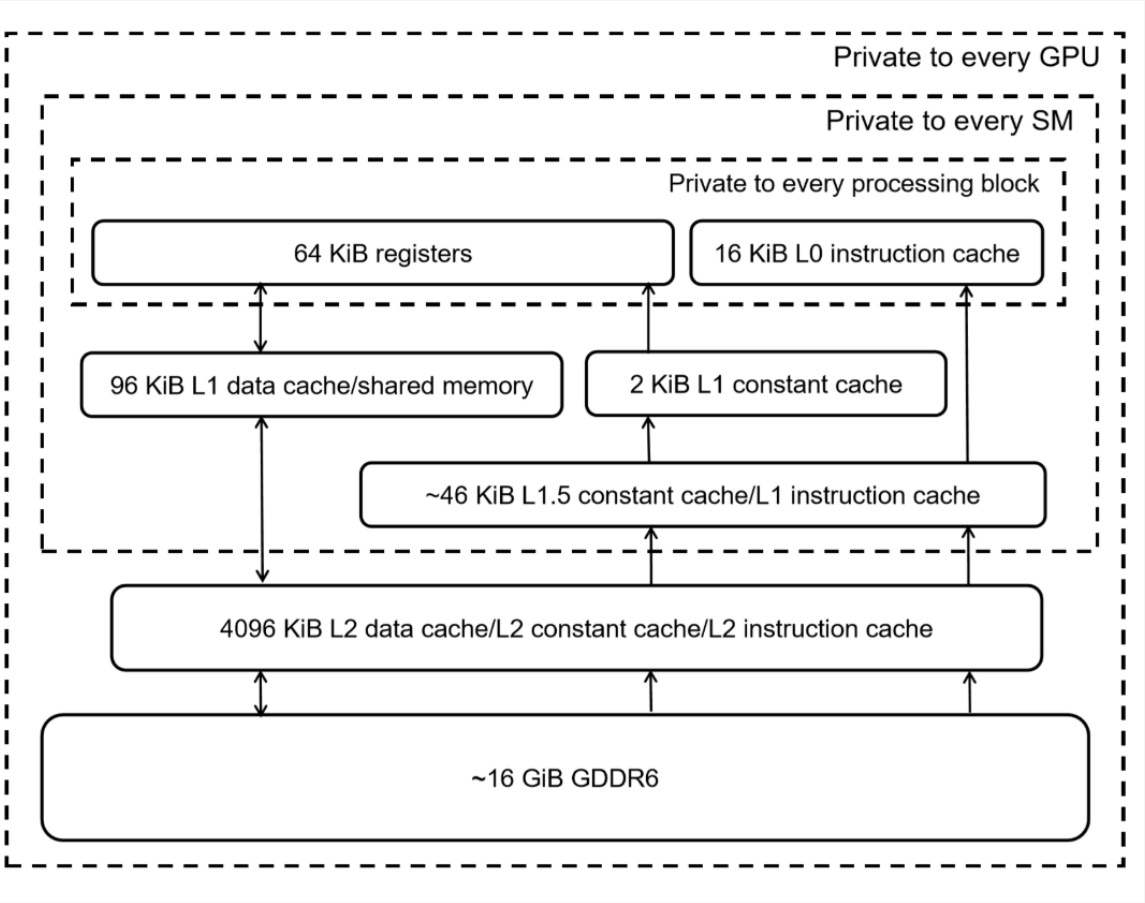


Image source: Dissecting the NVidia Turing T4 GPU via Microbenchmarking | Citadel Enterprise Americas LLC

CUDA Execution Model

Software

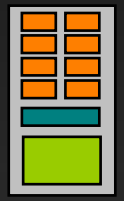
Thread

Thread Block

Grid

Hardware

CUDA core



Multiprocessor (SM)

Device

Threads are executed by CUDA cores

Thread blocks are executed on multiprocessors (SM)

- Thread blocks do not migrate
- Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)

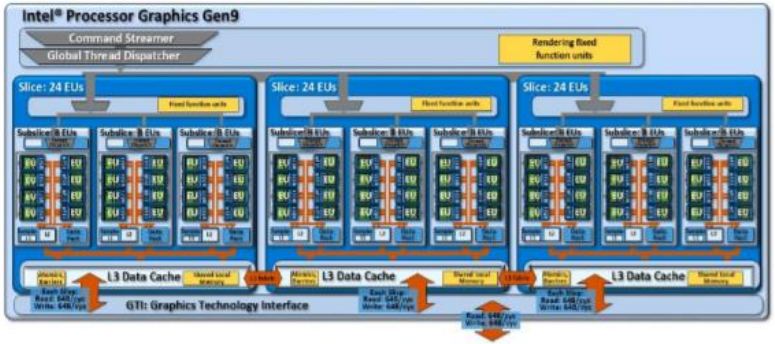
A kernel is launched as a grid of thread blocks

Blocks and grids can be multi dimensional (x,y,z)

Quantum of independence: Warps

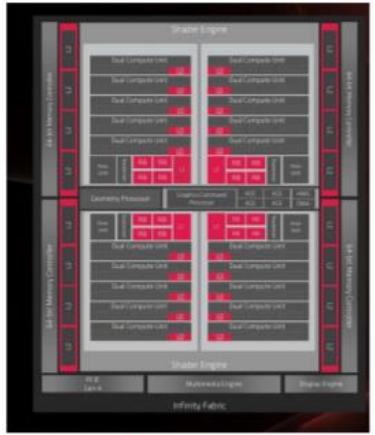
- A thread block consists of one or more warps
- A warp is executed physically in parallel (SIMD) on a multiprocessor
- The SM creates, manages, schedules and executes threads at warp granularity
- All threads in a warp execute the same instruction. If threads of a warp diverge the warp serially executes each branch path taken.
- When a warp executes an instruction that accesses global memory it coalesces the memory accesses of the threads within the warp into as few transactions as possible
- Currently all NVIDIA GPUs use a warp size of 32

But Every Generation and Manufacturer Is Different



Intel Gen9 GPU

NVIDIA V100



AMD

From Exascale Computing Program Annual Meeting: *SYCL Programming Model for Aurora*

And The Terminology Changes

Nvidia/CUDA Terminology	AMD Terminology	Intel Terminology	Description (pulled almost word-for-word from AMD slides)
Streaming Multiprocessor (SM)	Compute Unit (CU)	SubSlice (SS)	One of many independent parallel vector processors in a GPU that contain multiple SIMD ALUs.
Kernel	Kernel	Kernel	Functions launched to the GPU that are executed by multiple parallel workers on the GPU.
Warp	Wavefront	Vector thread, issuing SIMD instruction	Collection of operations that execute in lockstep, run the same instructions, and follow the same control-flow path. Individual lanes can be masked off.
Thread block	Workgroup	Workgroup	Group of warps/wavefronts/vector threads that are on the GPU at the same time. Can synchronize together and communicate through local memory.
Thread	Work item/Thread	Work item/Vector lane	Individual lane in a warp/wavefront/vector thread
Global Memory	Global Memory	GPU Memory	DRAM memory accessible by the GPU that goes through some layers of cache
Shared memory	Local memory	Shared local memory	Scratchpad that allows communication between warps/wavefronts/vector threads in a threadblock/workgroup
Local memory	Private memory	GPRF	Per-thread private memory, often mapped to registers.

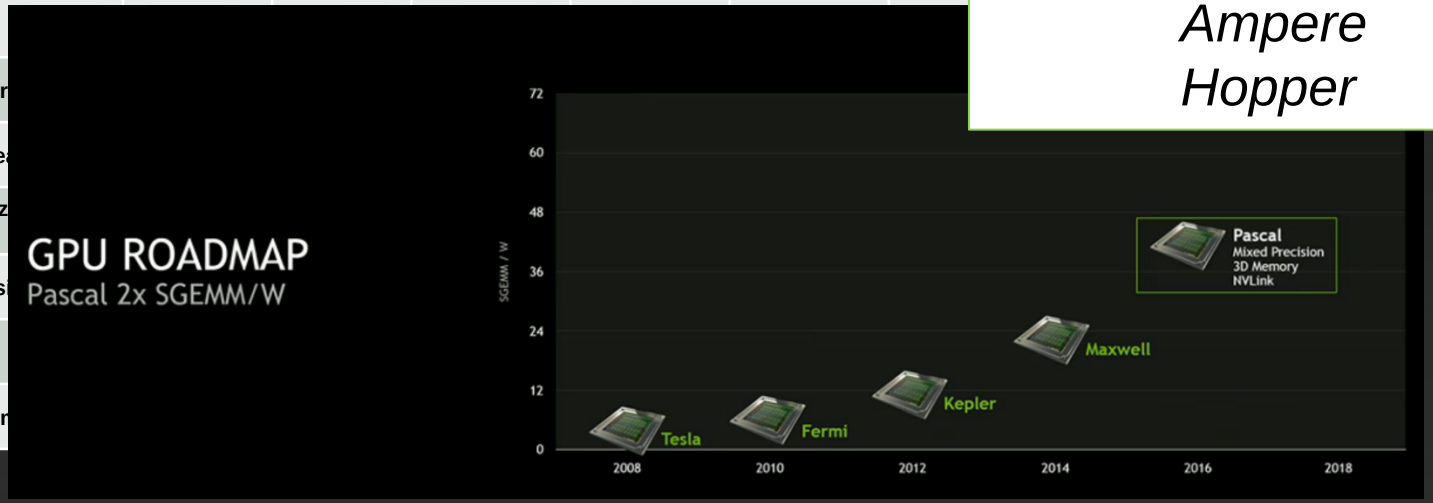
Rapid Evolution

	Fermi GF100	Fermi GF104	Kepler GK104	Kepler GK110	Maxwell GM107	Pascal GP100
Compute Capability	2.0	2.1	3.0	3.5	5.0	6.0
Threads / Warp	32	32	32	32	32	32
Max Warps / Multiprocessor	48	48	54	64	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048	2048	
Max Thread Blocks / Multiprocessor	8	8	16	16	32	
32-bit Registers / Multiprocessor						
Max Registers / Thr						
Max Threads / Thre						
Shared Memory Siz Configurations						
Max X Grid Dimensi						
Hyper-Q						
Dynamic Parallelisr						

- Do you want to have to keep up with this?
- Maybe the compiler knows more about this than you? Is that possible?

Evolution continues with:

Turing
Ampere
Hopper



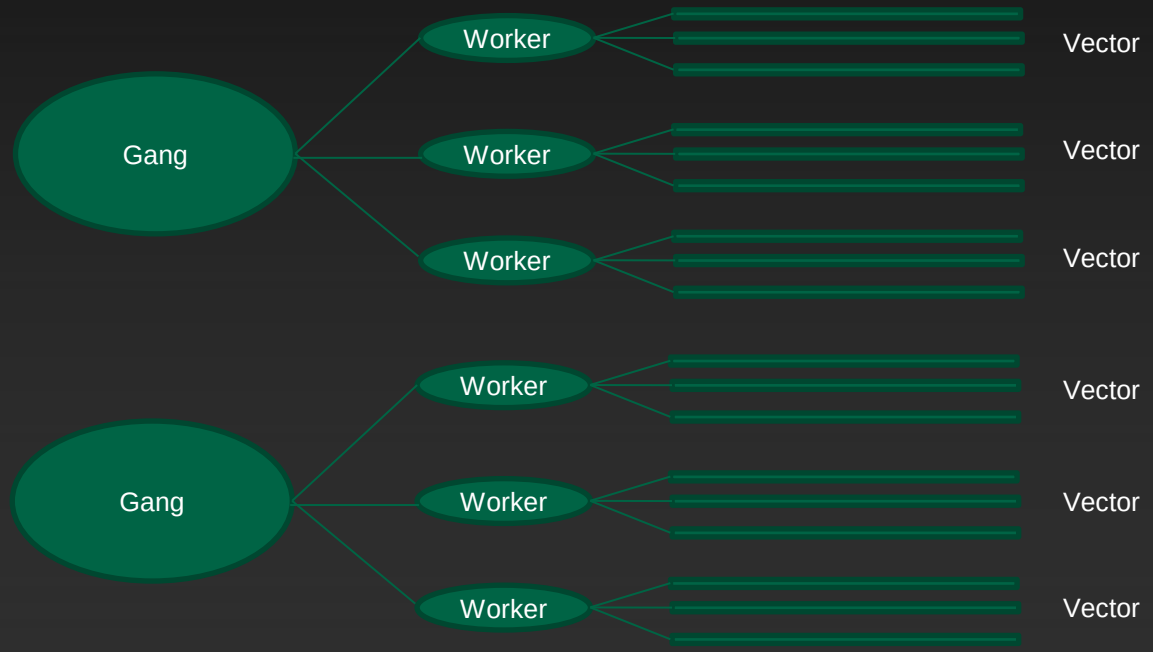
Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	NA	NA	8
Tensor Cores / GPU	NA	NA	NA	640
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1530 MHz
Peak FP32 TFLOP/s*	5.04	6.8	10.6	15.7
Peak FP64 TFLOP/s*	1.68	.21	5.3	7.8
Peak Tensor Core TFLOP/s*	NA	NA	NA	125
Texture Units	240	192	224	320
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB	6144 KB
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB	Configurable up to 96 KB
Register File Size / SM	256 KB	256 KB	256 KB	256KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB	20480 KB
TDP	235 Watts	250 Watts	300 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion	21.1 billion
GPU Die Size	551 mm²	601 mm²	610 mm²	815 mm²
Manufacturing Process	28 nm	28 nm	16 nm FinFET+	12 nm FFN

This is a *good* thing.

- Don't put yourself in a situation where this becomes a maintenance nightmare.
- Leave that problem to the compiler writers.

OpenACC Task Granularity

- The OpenACC execution model has three levels: *gang*, *worker* and *vector*
- This is supposed to map to any architecture that is a collection of Processing Elements (PEs) where each PE is multithreaded and each thread can execute vector instructions.



Targeting the Architecture

As we said, OpenACC assumes a device will contain multiple processing elements (PEs) that run in parallel. Each PE also has the ability to efficiently perform vector-like operations. For NVIDIA GPUs, it is reasonable to think of a PE as a streaming multiprocessor (SM). Then an OpenACC *gang* is a threadblock, a *worker* is effectively a warp, and an OpenACC *vector* is a CUDA thread. Phi, or similar Intel SMP architectures also map in a logical, but different, fashion.

	GPU	SMP (Phi)
Vector →	Thread →	SSE Vector
Worker →	Warp →	Core
Gang →	SM →	CPU

Kepler, for example

- Block Size Optimization:
  - 32 thread wide blocks are good for Kepler, since warps are allocated by row first.
    - 32 thread wide blocks will mean all threads in a warp are reading and writing contiguous pieces of memory
    - Coalescing
  - Try to keep total threads in a block to be a multiple of 32 if possible
    - Non-multiples of 32 waste some resources & cycles
  - Total number of threads in a block: between 256 and 512 is usually a good number.
- Grid Size Optimization:
  - Most people start with having each thread do one unit of work
  - Usually better to have fewer threads so that each thread could do multiple pieces of work.
  - What is the limit to how much smaller we can make the number of total blocks?
    - We still want to have at least as many threads as can fill the GPU many times over (for example 4 times). That means we need at least 2880 x 15 x 4 = ~173,000 threads
    - Experiment by decreasing the number of threads

Mapping OpenACC to CUDA Threads and Blocks

```
#pragma acc kernels
for( int i = 0; i < n; ++i )
      y[i] += a*x[i];
```

16 blocks, 256 threads each.

```
#pragma acc kernels loop gang(100) vector(128)
for( int i = 0; i < n; ++i )
      y[i] += a*x[i];
```

100 thread blocks, each with 128 threads, each thread executes one iteration of the loop.

```
#pragma acc parallel num_gangs(100) vector_length(128)
{
   #pragma acc loop gang vector
   for( int i = 0; i < n; ++i ) y[i] += a*x[i];
}
```

100 thread blocks, each with 128 threads, each thread executes one iteration of the loop, using parallel

SAXPY Returns For Some Fine Tuning

The default (will work OK):

```
#pragma acc kernels loop
for( int i = 0; i < n; ++i )
        y[i] += a*x[i];
```

Some suggestions to the compiler:

```
#pragma acc kernels loop gang(100), vector(128)
for( int i = 0; i < n; ++i )
         y[i] += a*x[i];
```

Specifies that the kernel will use 100 thread blocks, each with 128 threads, where each thread executes one iteration of the loop. This beat the default by ~20% *last time I tried…*

Parallel Regions vs. Kernels

We have been using *kernels* thus far, to great effect. However OpenACC allows us to very explicitly control the flow and allocation of tasks using *parallel* regions.

These approaches come from different backgrounds.

PGI Accelerator *Region**	→	OpenACC *kernels*
OpenMP *parallel*	→	OpenACC *parallel*

*Similar philosophy to preferring OpenMP *omp parallel for*

Parallel Regions

When you start an accelerator parallel region, one or more gangs of workers are created to execute the accelerator parallel region. The number of gangs, and the number of workers in each gang and the number of vector lanes per worker remain constant for the duration of that parallel region.

Each gang begins executing the code in the structured block in gang-redundant mode. This means that code within the parallel region, but outside of a loop construct with gang-level worksharing, will be executed *redundantly* by all gangs. One worker in each gang begins executing the code in the structured block of the construct.

This means you are setting the cores free, allowing them to attack the work with maximum efficiency. Now it is up to you to coral them into some kind of sensible activity!

Parallel Construct

Fortran

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

C

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

Clauses

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

```
private( list )
firstprivate( list )
reduction( operator:list )
.
.
```

Also any data clause

Parallel Clauses

num_gangs(expression)	Controls how many parallel gangs are created.
num_workers(expression)	Controls how many workers are created in each gang.
vector_length(list)	Controls vector length of each worker.
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.
copy(),copyin(),copyout(),create()	Same behavior we already know.
present(list)	Variable already there from some other data clause. Suppress any desire of compiler to copy and do nothing.
asynch()/wait()	Just getting to this in a few slides...

Parallel Regions

As in OpenMP, the OpenACC parallel construct creates a number of parallel gangs that immediately begin executing the body of the construct *redundantly*. When a gang reaches a work-sharing loop, that gang will execute a subset of the loop iterations. One major difference between the OpenACC parallel construct and OpenMP is that there is no barrier at the end of a work-sharing loop in a parallel construct.

SAXPY as a parallel region

```
#pragma acc parallel num_gangs(100), vector_length(128)
{
#pragma acc loop gang, vector
for( int i = 0; i < n; ++i )
        y[i] = y[i] + a*x[i];
}
```

Compare and Contrast

Let's look at how this plays out in actual code.

This

```
#pragma acc kernels
{
        for( i = 0; i < n; ++i )
                a[i] = b[i] + c[i];
}
```

Is the same as

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

Don’t Do This

But not

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

By leaving out the loop directive, we get totally redundant execution of the loop by each gang. This is not desirable, to say the least.

Parallel Regions vs. Kernels

From these simple examples you could get the impression that simply putting in loop directives everywhere would make parallel regions equivalent to kernels. That is not the case.

The sequence of loops here

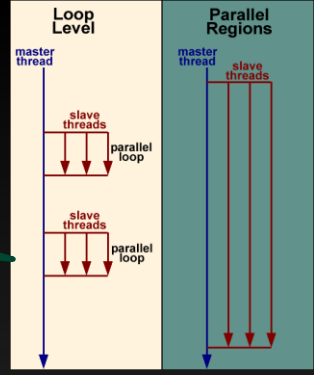
```
#pragma acc kernels
{
for (i=0; i<n; i++)
        a(i) = b(i)*c(i)
for (i=1; i<n-1; i++)
        d(i) = a(i-1) + a(i+1)
}
```

does what you might think. Two kernels are generated and the first completes before the second starts.

A parallel region will work differently

```
#pragma acc parallel
{
#pragma acc loop
for (i=0; i<n; i++)
        a(i) = b(i)*c(i)
#pragma acc loop
for (i=1; i<n-1; i++)
        d(i) = a(i-1) + a(i+1)
}
```

Straight from the pages of our OpenMP lecture!



The compiler will start some number of gangs and then work-share the iterations of the first loop across those gangs, and work-share the iterations of the second loop across the same gangs. There is no synchronization between the first and second loop, so there's no guarantee that the assignment to a(i) from the first loop will be complete before its value is fetched by some other gang for the assignment in the second loop. This will result in incorrect results.

But the most common reason we use parallel regions is because we want to eliminate these wasted blocking cycles. So we just need some means of controlling them…

Controlling Waiting

We can allow workers, or our CPU, to continue ahead while a loop is executing as we wait at the appropriate times (so we don't get ahead of our data arriving or a calculation finishing. We do this with **asynch** and **wait** statements.

```
#pragma acc parallel loop async(1)
for (i = 0; i < n; ++i)
  c[i] += a[i];
#pragma acc parallel loop async(2)
for (i = 0; i < n; ++i)
  b[i] = expf(b[i]);
#pragma acc wait
// host waits here for all async activities to complete
```

We are combining the parallel region and loop directives together. A common idiom.

Note that there is no sync available *within* a parallel region (or kernel)!

Using Separate Queues

We have up to 16 queues that we can use to manage completion dependencies.

```
#pragma acc parallel loop async(1)          // on queue 1
for (i = 0; i < n; ++i)
  c[i] += a[i];
#pragma acc parallel loop async(2)          // on queue 2
for (i = 0; i < n; ++i)
  b[i] = expf(b[i]);
#pragma acc parallel loop async(1) wait(2)  // waits for both
for (i = 0; i < n; ++i)
  d[i] = c[i] + b[i];
// host continues executing while GPU is busy
```

Dependencies

We can use these with kernels too.

```
#pragma acc kernels loop independent async(1)
for (i = 1; i < n-1; ++i) {
  #pragma acc cache(b[i-1:3], c[i-1:3])
  a[i] = c[i-1]*b[i+1] + c[i]*b[i] + c[i+1]*b[i-1];
}
#pragma acc parallel loop async(2) wait(1)   // start queue 2
for (i = 0; i < n; ++i)                      // after 1
  c[i] += a[i];                              // need a to finish
#pragma acc parallel loop async(3) wait(1)   // start queue 3
for (i = 0; i < n; ++i)                      // after 1
  b[i] = expf(b[i]);                         // don't mess with b
// host continues executing while GPU is busy
```

Private Variables

One other important consideration for parallel regions is what happens with scaler (non-array) variables inside loops. Unlike arrays, which are divided up amongst the cores, the variables are shared by default. This is often not what you want.

If you have a scaler inside a parallel loop that is being changed, you probably want each core to have a ***private*** copy. This is similar to what we saw earlier with a reduction variable.

```
integer nsteps, i
double precision step, sum, x
nsteps = ...
sum = 0
step = 1.0d0 / nsteps
!$acc parallel loop private(x) reduction(+:sum)
do i = 1, nsteps
  x = (i + 0.5d0) * step
  sum = sum + 1.0 / (1.0 + x*x)
enddo
pi = 4.0 * step * sum
```

Consistent with this philosophy, scaler variables default to ***firstprivate*** inside of ***parallel*** regions where ***kernel*** regions default to ***copy***. Both regions default to ***copy*** for aggregate types.

Loop Clauses

`private (list)`	Each thread gets it own copy (implied for index variable).
`reduction (operator:list)`	Also private, but combine at end. Your responsibility now!
`gang/worker/vector( )`	We've seen these.
`independent`	Independent. Ignore any suspicions.
`seq`	Opposite. Sequential, don't parallelize.
`auto`	Compiler's call.
`collapse()`	Says how many nested levels apply to this loop. Unrolls. Good for small inner loops.
`tile(,)`	Opposite. Splits each specified level of nested loop into two. Good for locality.
`device_type()`	For multiple devices.

Kernels vs. Parallel

Advantages of kernels

- compiler autoparallelizes
- best with nested loops and no procedure calls
- one construct around many loop nests can create many device kernels

Advantages of parallel

- some compilers are bad at parallelization
- more user control, esp. with procedure calls
- one construct generates one device kernel
- similar to OpenMP

Parallel Regions vs. Kernels (Which is best?)

To put it simply, kernels leave more decision making up to the compiler. There is nothing wrong with trusting the compiler ("trust but verify"), and that is probably a reasonable place to start.

If you are an OpenMP programmer, you will notice a strong similarity between the tradeoffs of kernels and regions and that of OpenMP parallel for/do versus parallel regions. We will discuss this later when we talk about OpenMP 4.0.

As you gain experience, you may find that the parallel construct allows you to apply your understanding more explicitly. On the other hand, as the compilers mature, they will also be smarter about just doing the right thing. *History tends to favor this second path heavily.*

Data Management

Again, as you get farther from a simple program, you may find yourself needing to manage data transfers in a more explicit manner. We restricted ourselves to the data copy type commands for our initial work, but still found update to be necessary. In general you won't find yourself frustrated for lack of a convenient data movement action.

enter data	Like *copyin* except that they do not need to apply to a structured block or scope. Could just stick one in some initialization routine. Clauses can be *async*, *wait*, *copyin* or *create*.
exit data	Bookend of above, but in addition to *async* and *wait* has *copyout*, and *delete* (decrement reference count) and finalize (force count to zero).
update	As used earlier, but has *async*, *wait* and some other clauses.

Dynamic Memory

You may have wondered how these data transfers cope with dynamic memory. The answer is, very naturally as OpenACC is intended for serious codes which usually use dynamic allocation. Here is one way that you might find yourself allocating/deallocating a dynamic structure on both the host and device.

C

```
tmp = (double *) malloc(count*sizeof(double));
#pragma acc enter data create(tmp[0:count])
.
.
.
#pragma acc exit data delete(tmp)
free(tmp)
```

Fortran

```
allocate(tmp(count))
!$acc enter data create(tmp)
.
.
.
!$acc exit data delete(tmp)
deallocate(tmp)
```

Declare Directive

You can put your data movement specification close to your natural variable declarations.

declare create	create on host and device, you will probably use *update* to manage
declare device_resident	create on device only, only accessible in compute regions
declare link and *declare create* (pointer)	pointers are created for data to be copied

Data Structures

Somebody has probably asked this by now, but if not, it is important for me to note that complex data structures are just fine for OpenACC. Feel free to use:

- Complex structs
- C++ classes
- Fortran derived types
- Dynamically allocated memory
- STL? Yes and No

The major caveat is that pointer based structures will not naturally move from CPU to GPU. This should be no surprise. You must do you own “deep copy” if you need to move such data.

Cache Directive

CUDA programmers always want to know how to access CUDA shared memory. All of you should be interested in how you can utilize this small (~48KB) shared (by the gang) memory for items that should be kept close at hand.

```
real temp(64)

!$acc parallel loop gang vector_length(64) private(temp)
do i = 1, n
  !$acc cache(temp)
  !$acc loop vector
  do j = 1, 64
    temp(j) = a(i,j)
    ....
```

CUDA Unified Memory*

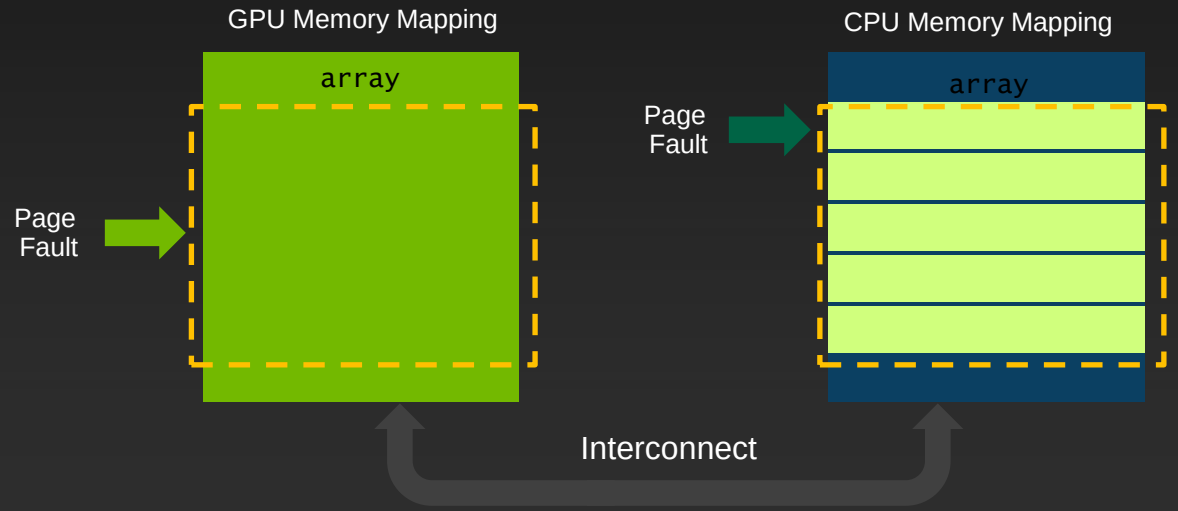
Speaking of memory, a few realistic words are in order concerning the awesome sounding Unified Memory. No more data management?

GPU Code

```
__global__
Void setValue(char *ptr, int index, char val)
{
  ptr[index] = val;
}
```

CPU Code

```
cudaMallocManaged(&array, size);
memset(array, size);
setValue<<<...>>>(array, size/2, 5);
```



* "*CUDA 8 and Beyond*", Mark Harris, GPU Technology Conference, April 4-7, 2016

OpenACC 2.0, 2.5, 2.7 & 3.0

Things you didn't know were missing.

The latest versions of the specification have a lot of improvements. The most anticipated ones remove limitations that you, as new users, might not have known about.

- Procedure Calls
- Nested Parallelism

As well as some other things that you might not have thought about

- Device specific tuning
- Multiple host thread support

Don't be afraid to review the full spec at

https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.7.pdf

Procedure Calls

In OpenACC 1.0, all procedures had to be inlined. This limitation has been removed, but you do need to follow some rules.

```
#pragma acc routine worker
extern void solver(float* x, int n);
.
.
.
#pragma acc parallel loop num_gangs(200)
for( int index = 0; index < N; index++ ){
          solver( X, n);
          .
          .
}
```

```
#pragma acc routine worker
void solver(float* x, int n){
.
.
          #pragma acc loop
          for( int index = 0; index < n; index++ ){
                    x[index] = x[index+2] * alpha;
                    .
                    .
          }
          .
}
```

In this case, the directive tells the compiler that “solver” will be a device executable and that it may have a loop at the worker level. No caller can do worker level parallelism.

Nested Parallelism

The previous example had gangs invoking workers. But it is now possible to have kernels actually launch new kernels.

```
#pragma acc routine
extern void solver(float* x, int n);
.
.
#pragma acc parallel loop
for( int index = 0; index < N; index++ ){
          solver( X, index);
}
```

```
#pragma acc routine
void solver(float* x, int n){
          #pragma acc parallel loop
          for( int index = 0; index < n; index++ ){
                    x[index] = x[index+2] * alpha
                    .
                    .
          }
          .
}
```

Having thousands of lightweight threads launching lightweight threads is probably not the most likely scenario.

Nested Parallelism

This is a more useful case. We have a single thread on the device launching parallelism from its thread.

```
#pragma acc routine
extern void solver(float* x, int n);
.
.
#pragma acc parallel num_gangs(1)
{
          solver( X, n1 );
          solver( Y, n2 );
          solver( Z, n3 );
}
```

```
#pragma acc routine
void solver(float* x, int n){
          #pragma acc parallel loop
          for( int index = 0; index < n; index++){
                    x[index] = x[index+2] * alpha;
                    .
                    .
          }
          .
}
```

The objective is to move as much of the application to the accelerator and minimize communication between it and the host.

Device Specific Tuning

I hope from our brief detour into GPU hardware specifics that you have some understanding of how hardware specific these optimizations can be. Maybe one more reason to let kernel do its thing. However, OpenACC does have ways to allow you to account for various hardware details. The most direct is device_type().

```
#pragma acc parallel loop  device_type(nvidia) num_gangs(200) \
                           device_type(radeon) num_gangs(800)
for( index = 0; index < n; index++ ){
        x[i] += y[i];
        solver( x, y, n );
}
```

Line continuation syntax
(in case we haven't see this yet)

Multiple Devices and Multiple Threads

- Multiple threads and one device: fine. You are responsible for making sure that the data is on the multi-core host when it needs to be, and on the accelerator when it needs to be there. But, you have those data clauses in hand already (present_or_copy will be crucial), and OpenMP has its necessary synchronization ability.

- Multiple threads and multiple devices. One might hope that the compilers will eventually make this transparent (i.e. logically merge devices into one), but at the moment you need to:
  - Assign threads to devices:
    - omp_get_thread_num
    - call acc_set_device_num
  - Manually break up the data structure into several pieces:
    - `!$acc kernels loop copyin(x(offs(i)+1:offs(i)+nsec),y(offs(i)+1:offs(i)+nsec))`
    - From excellent example on Page 25 of the PGI 2016 OpenACC Getting Started Guide

Profiling

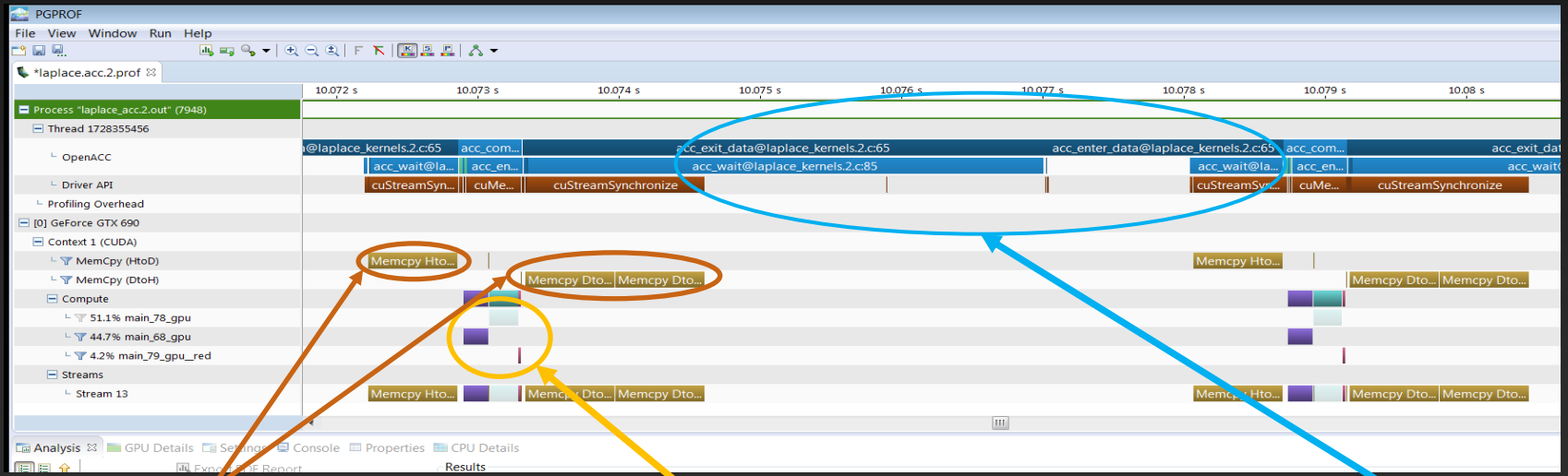
So, how do you recognize these problems (opportunities!) besides the relatively simple timing output we have used in this class?

One benefit of the NVIDIA ecosystem is the large number of tools from the CUDA community that we get to piggyback upon.

The following uses the NVIDIA Visual Profiler which is part of the CUDA Toolkit.

Visual Profiler and our Laplace first attempt.

- **We zoom in to get a better view of the timeline. As expected, it looks like our program is spending a significant amount of time transferring data between the host and device. We also see that the compute regions are very small, with a lot of distance between them.**



Device Memory Transfers

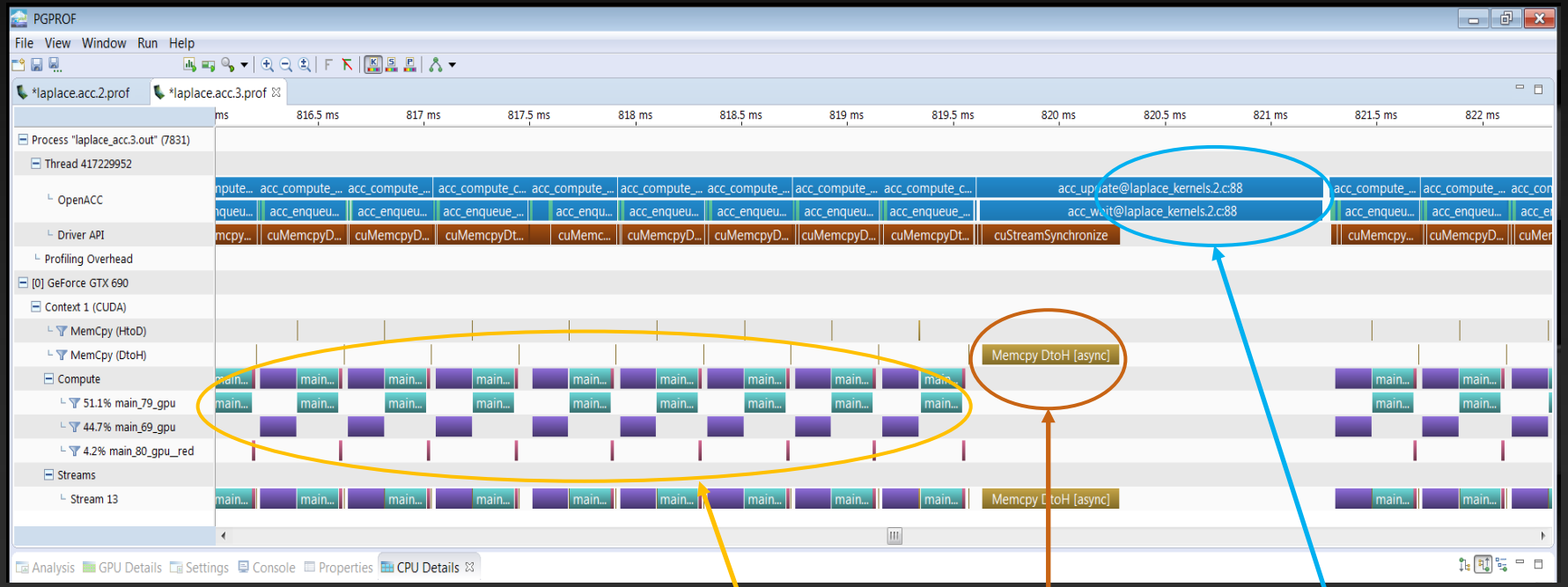
Compute on the Device

CPU overhead time including:
- Device data creation and deletion
- Host Memory Copy

PITTSBURGH SUPERCOMPUTING CENTER

After our data management fix.

After adding the data region, we see lots of compute and data movement only when we update.

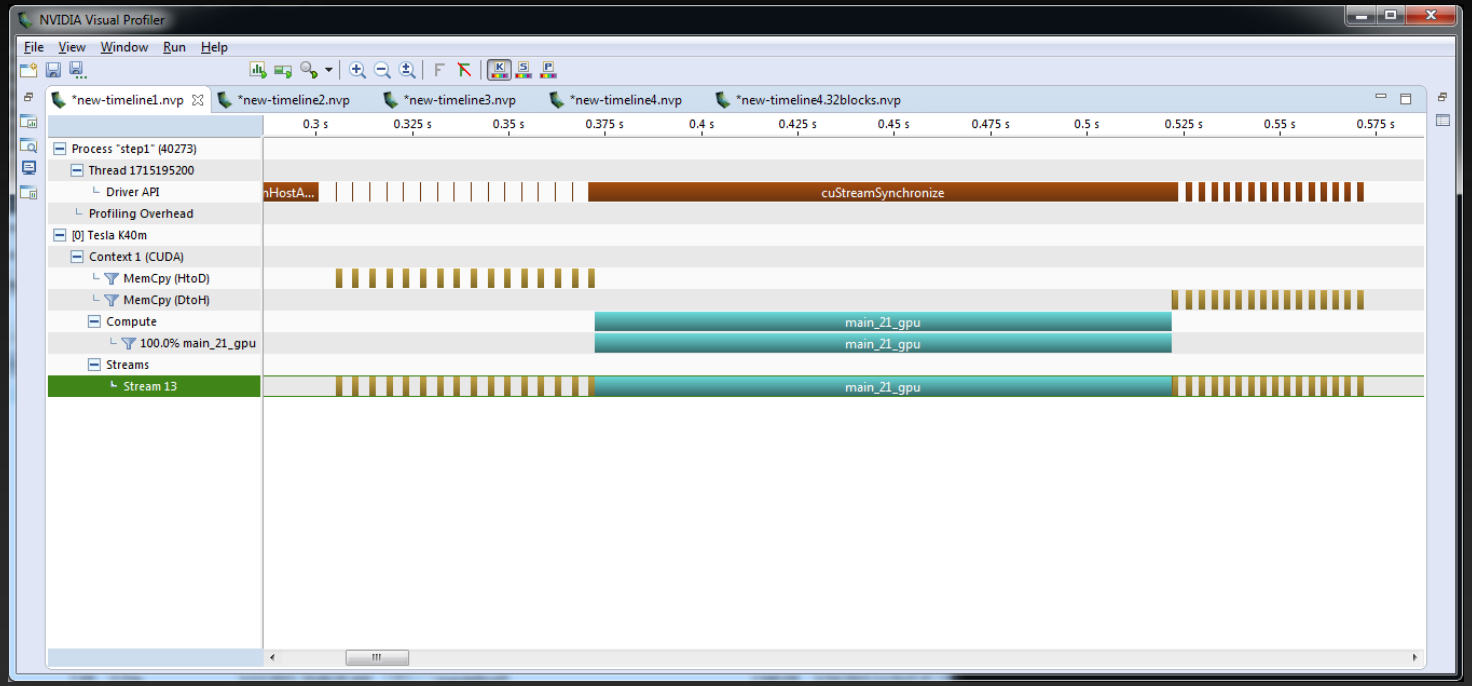


Device compute

Update to host

Host memory copy

Mandlebrot Code



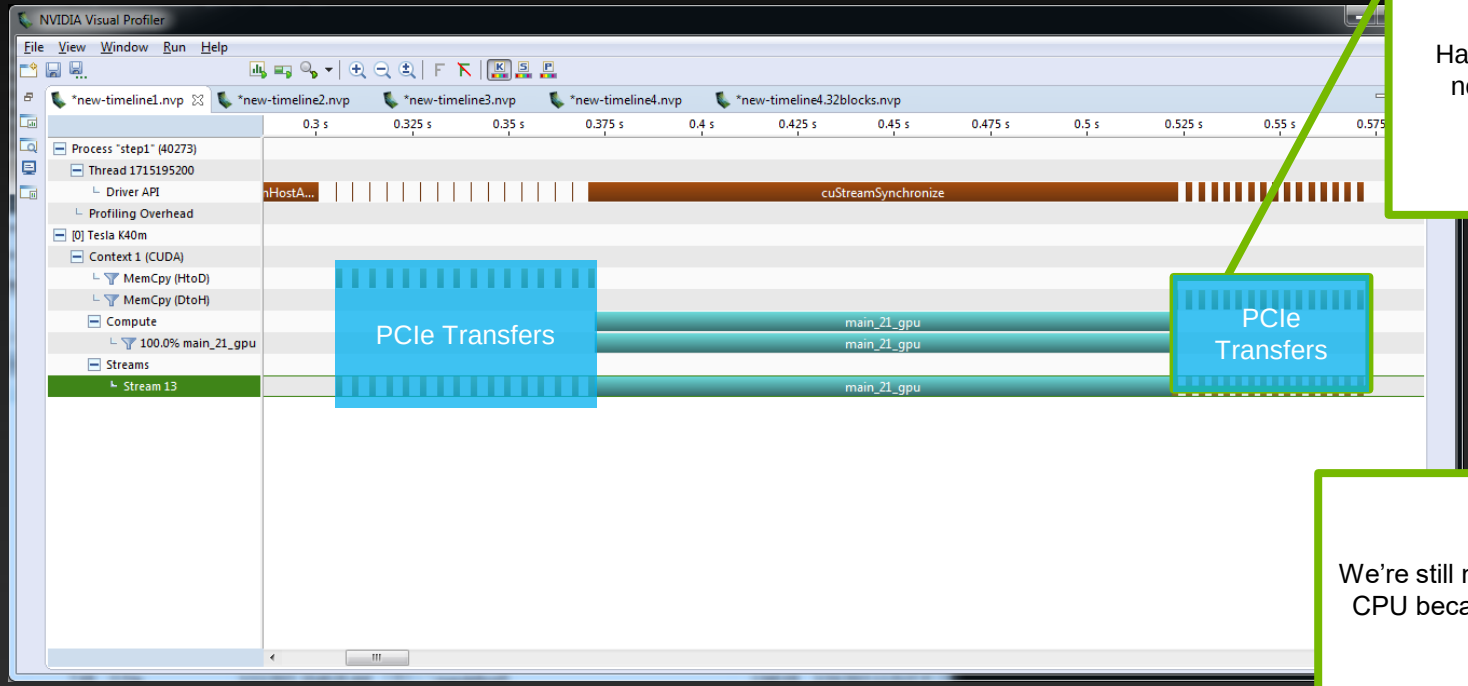
This is for an OpenACC Mandlebrot set image generation code from NVIDIA . You can grab it at

https://github.com/NVIDIA-OpenACC-Course/nvidia-openacc-course-sources

PITTSBURGH SUPERCOMPUTING CENTER

Lots of Data Transfer Time

Step 1 Profile

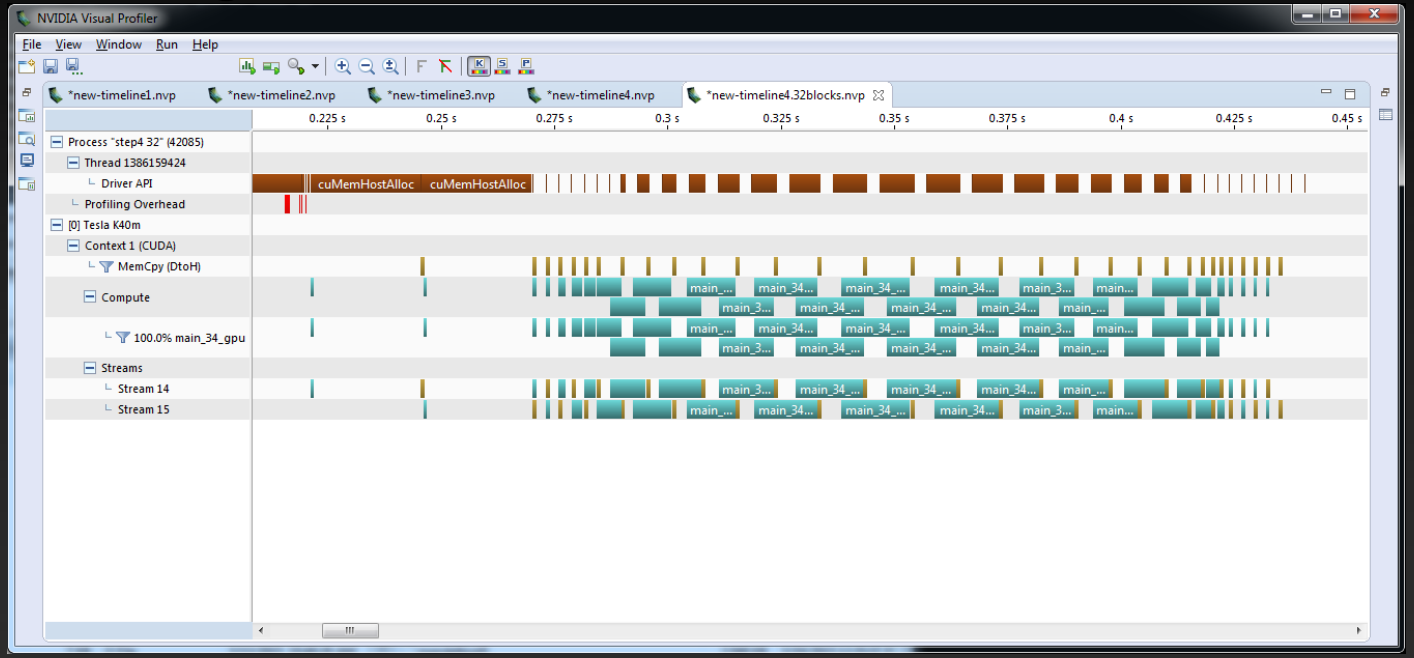


Half of our time is copying, none of it is overlapped.

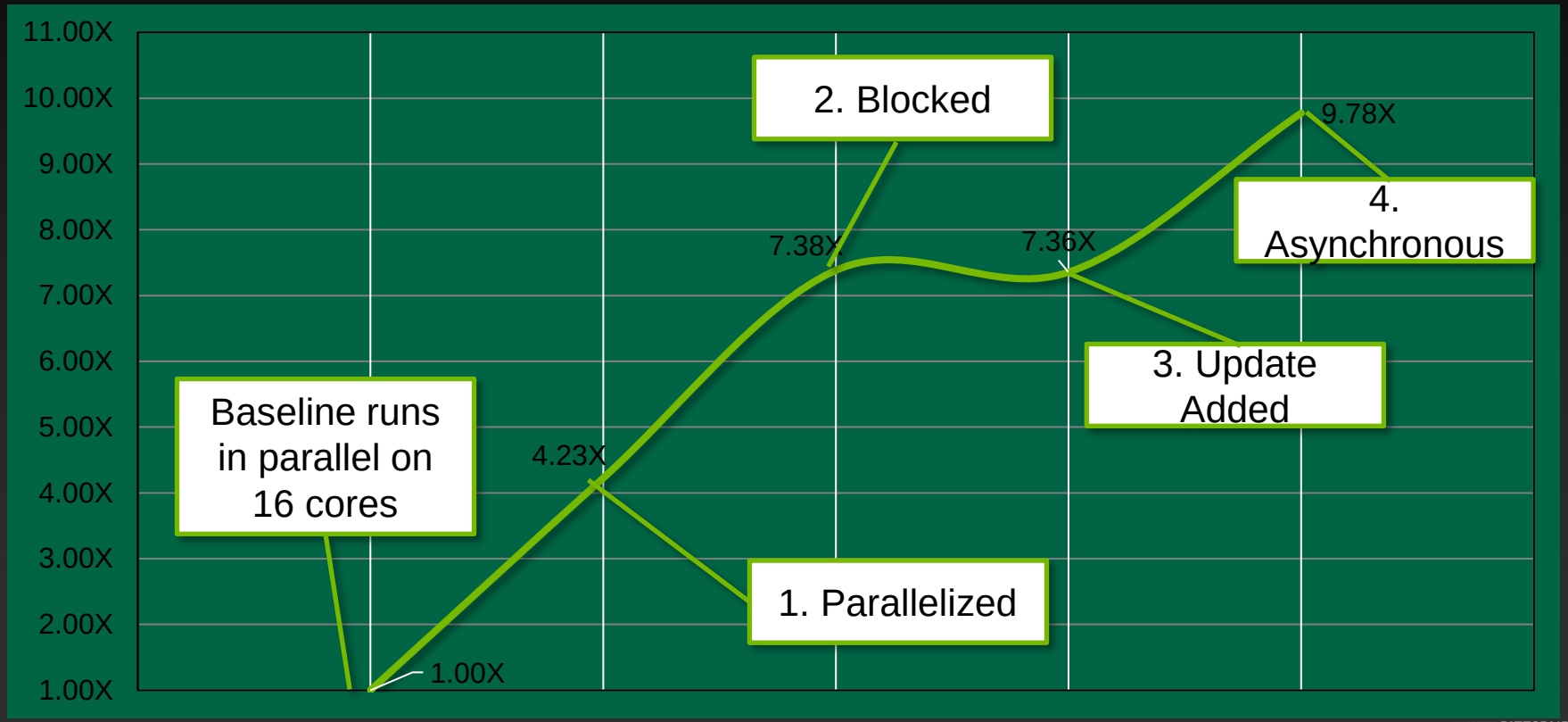
We're still much faster than the CPU because there's a lot of work.

Broken Into Blocks With Asynchronous Transfers

Pipelining with 32 blocks



Optimized In A Few Well-Informed Stages



OpenACC Things Not Covered

The OpenACC specification has grown quite accommodating as of Version 2.5. You have already seen some redundancy between directives, clauses and APIs, so I have made no attempt to do "laundry lists" of every option along the way. It would be quite repetitive. I think you are well prepared to glance at the OpenACC Specification and grasp just about all of it.

We have omitted various and sundry peripheral items. Without attempting to be comprehensive, here are a few topics of potential interest to some of you.

- Language specific features: C dynamic arrays, C++ classes, Fortran derived types. These particular items are well supported. An excellent guide to this topic is the PGI OpenACC Getting Started Guide (http://www.pgroup.com/doc/openacc_gs.pdf).
- Environment variables: useful for different hardware configurations
- if clauses, macros and conditional compilation: allow both runtime and compile time control over host or device control flow.
- API versions of nearly all directives and clauses
- Hybrid programming. *Works great!* Don't know how meaningful this is to you…

Should I Learn CUDA?

The situation today has a very similar historical precedent. Namely the evolution away from machine languages ("assembly") to C. To use PCs as a particular example.

1980's	1990's
DOS (Machine Language)	Windows, Linux (C)
Games (Machine Language)	Games (C)
Desktop Apps (C, Pascal, Basic)	Desktop Apps (C, C++, VB)

So, the answer is increasingly "probably not". I will guess most of you fall on the "no" side. Just like ML, you aren't really an "expert" unless you do understand what the compiler is doing with your high level approach, but that may not be necessary for your purposes.

A very important principle that remains valid is that any performant approach must allow you to understand what you are ultimately asking the hardware to do at the low level. A programmer that knows assembly knows fairly well what the C statement

$$X = X + 1$$

will become in ML: It will take a some cycles to fetch a value from memory/cache into a register and add a 1 to it. If you know how Python works, then you know that this same instruction might well take many hundreds of cycles, and it may be impossible to tell. If you know C++, this same line might generate exactly the same instructions as C, or it might involve an object and take a thousands of instructions (although you can almost always tell by closer inspection with C++).

Credits

Some of these examples are derived from excellent explanations by these gentlemen, and more than a little benefit was derived from their expertise.

Michael Wolfe, PGI

Jeff Larkin, NVIDIA

Mark Harris, NVIDIA

Cliff Woolley, NVIDIA

OpenMP and GPUs

John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center

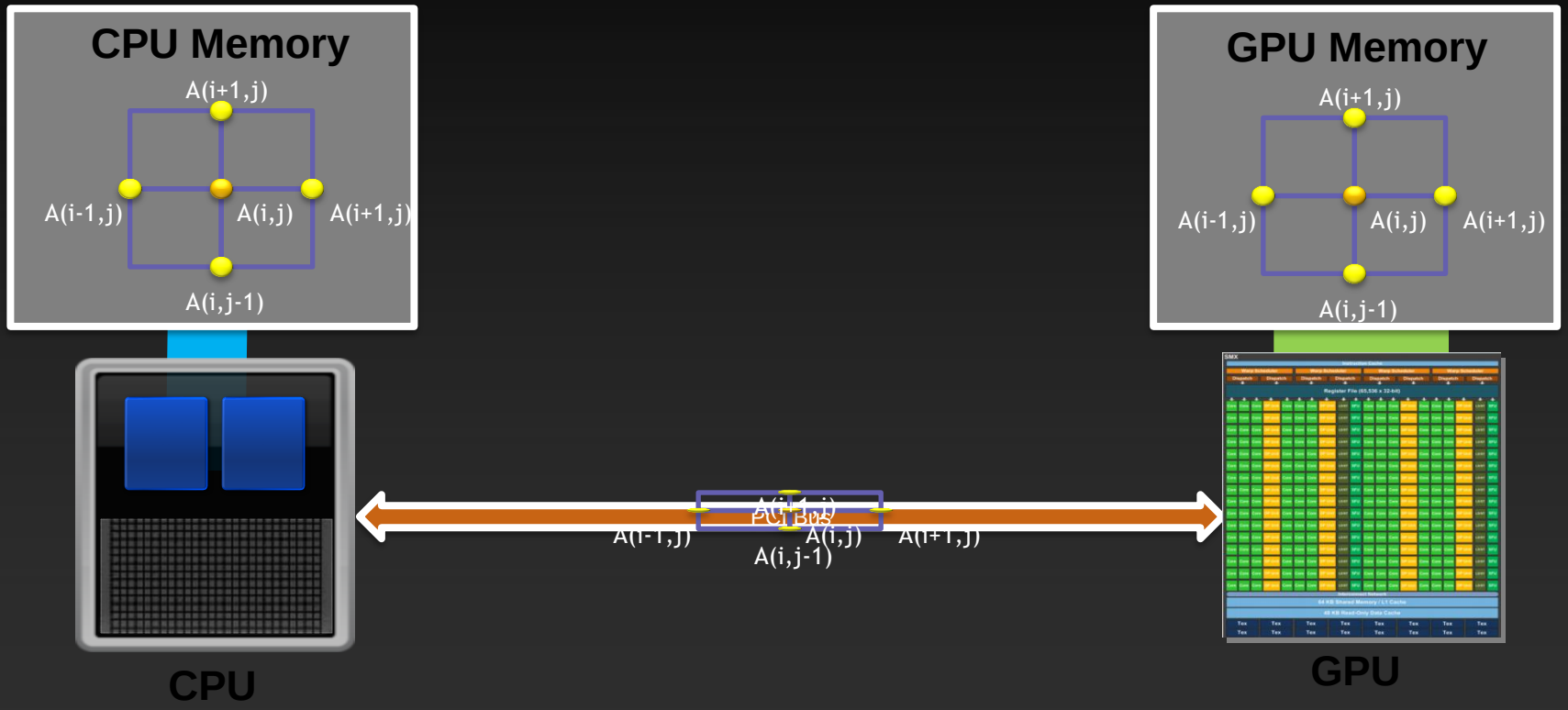
Classic OpenMP

OpenMP was designed to replace low-level and tedious multi-threaded programming solutions like POSIX threads, or Pthreads.

OpenMP was originally targeted towards controlling capable and completely independent processors, with shared memory. The most common such configurations today are the many multi-cored chips we all use. You might have dozens of threads, each of which takes some time to start or complete.

In return for the flexibility to use those processors to their fullest extent, OpenMP assumes that you know what you are doing. You prescribe what how you want the threads to behave and the compiler faithfully carries it out.

Then Came This



GPUs are not CPUs

GPU require memory management. We do not simply have a single shared dataspace.

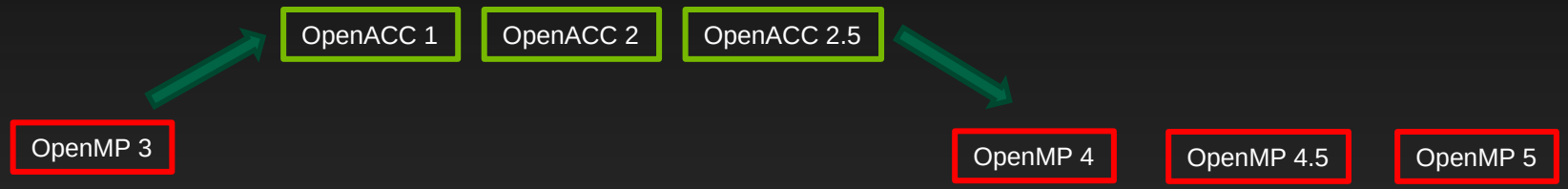
GPUs have thousands of cores.

But they aren't independent.

And they can launch very lightweight threads.

But it seems like the OpenMP approach provides a good starting point to get away from the low-level and tedious CUDA API...

Original Intention



Let OpenACC evolve rapidly without disturbing the mature OpenMP standard.
They can merge somewhere around version 4.0.

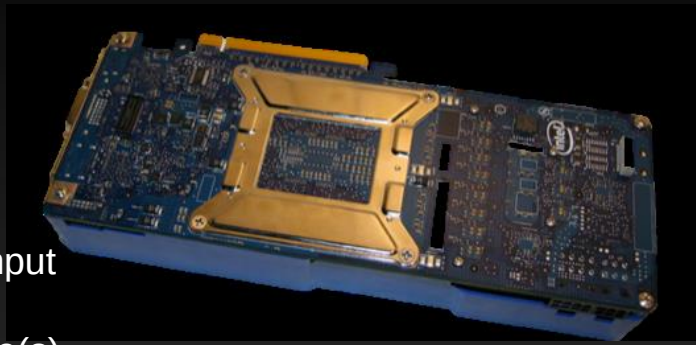
Meanwhile...

Since the days of RISC vs. CISC, Intel has mastered the art of figuring out what is important about a new processing technology and saying “why can’t we do this in x86?”

The Intel Many Integrated Core (MIC) architecture is about large die, simpler circuit, and much more parallelism, in the x86 line.

Courtesy Dan Stanzione, TACC

What ~~is~~ was MIC?



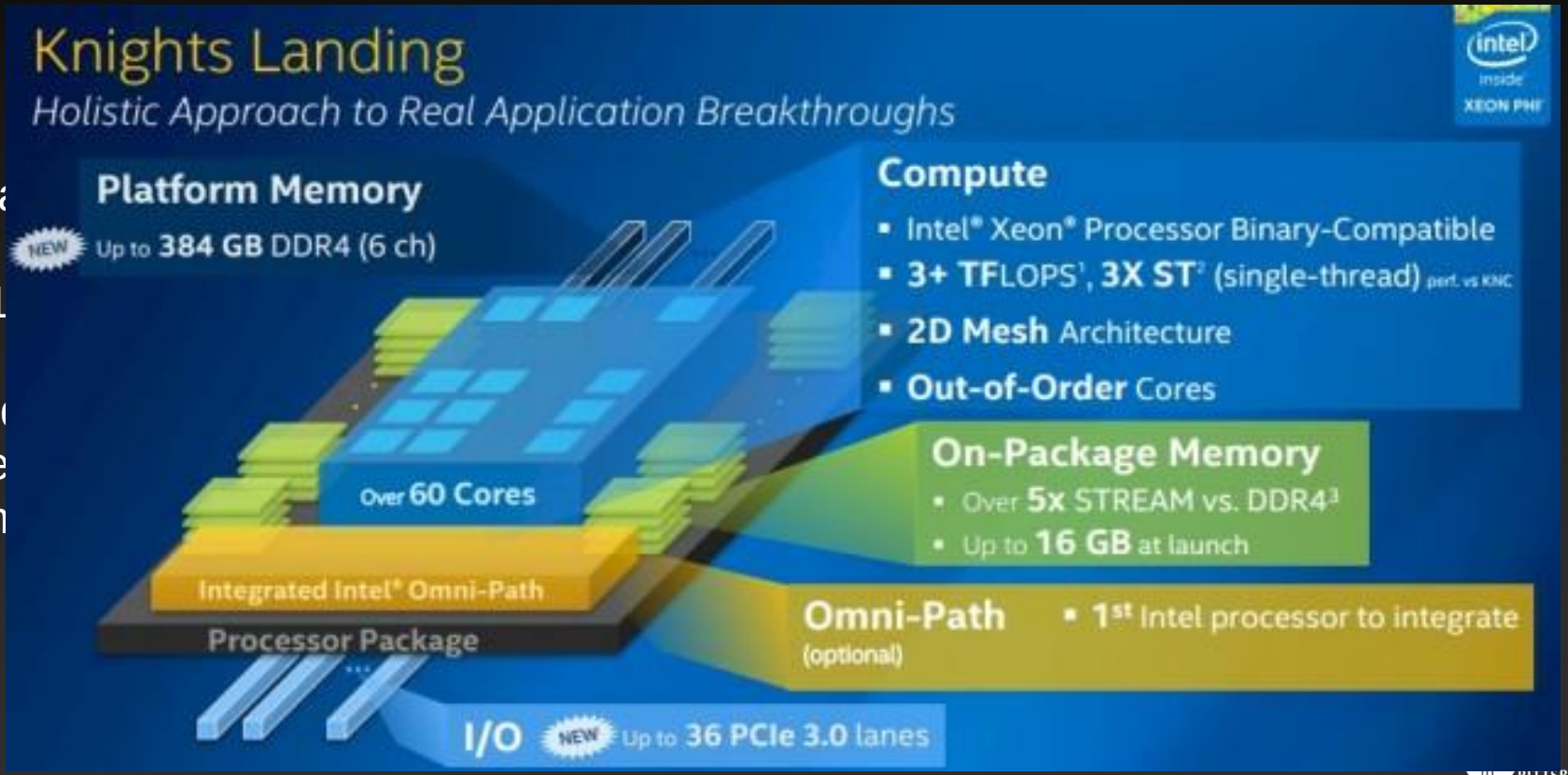
Basic Design Ideas:

- Leverage x86 architecture (a CPU with many cores)
- Use x86 cores that are simpler, but allow for more compute throughput
- Leverage existing x86 programming models
- Dedicate much of the silicon to floating point ops., keep some cache(s)
- Keep cache-coherency protocol
- Increase floating-point throughput per core
- Implement as a separate device
- Strip expensive features (out-of-order execution, branch prediction, etc.)
- Widened SIMD registers for more throughput (512 bit)
- Fast (GDDR5) memory on card

Courtesy Dan Stanzione, TACC

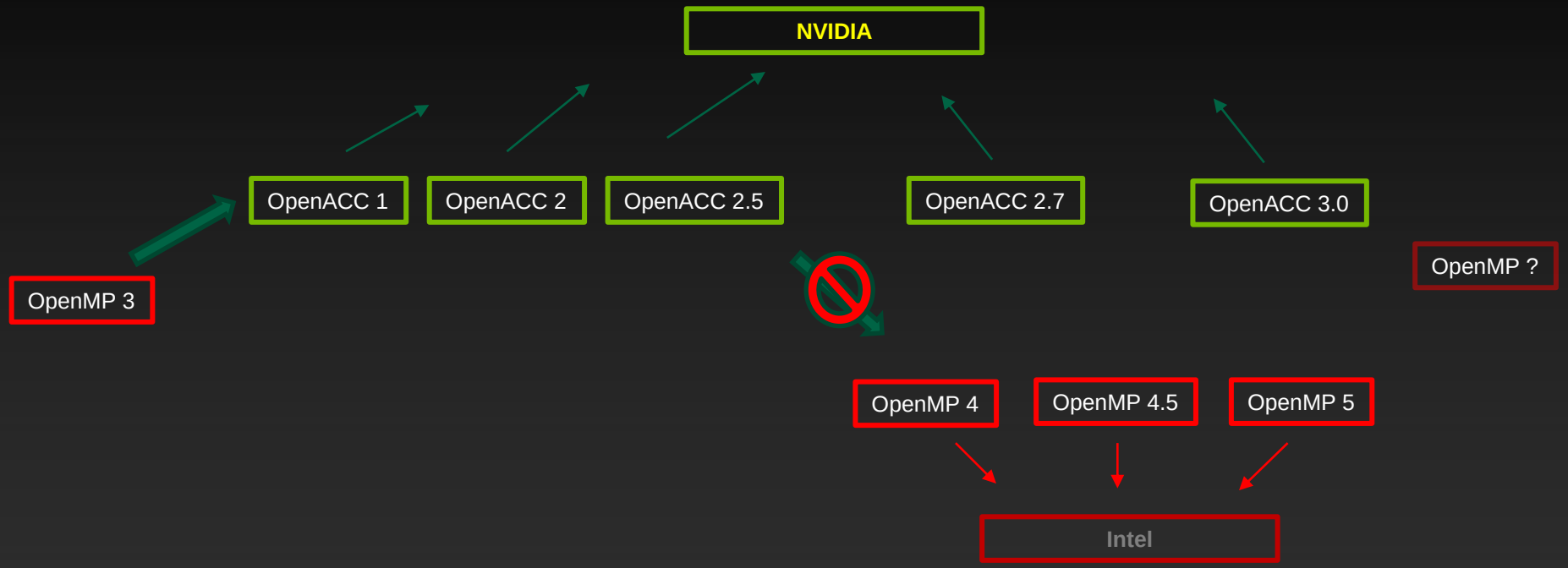
~~Latest~~ last MIC Architecture

- Ma
- L1
- Bi
  ne
  an



Courtesy Dan Stanzione, TACC

Implications for the OpenMP/OpenACC Merge



Intel and NVIDIA have both influenced their favored approached to make them more amenable to their own devices.

OpenMP 4.0

The OpenMP 4.0 standard did incorporate the features needed for accelerators, with an emphasis on Intel-like devices. We are left with some differences.

OpenMP takes its traditional prescriptive approach ("*this is what I want you to do*"), while OpenACC could afford to start with a more modern (compilers are smarter) descriptive approach ("*here is some parallelizable stuff, do something smart*"). This is practically visible in such things as OpenMP's insistence that you identify loop dependencies, versus OpenACC's *kernel* directive, and its ability to spot them for you.

OpenMP assumes that every thread has its own synchronization control (`barriers`, `locks`), because real processors can do whatever they want, whenever. GPUs do not have that at all levels. For example, NVIDIA GPUs have synchronization at the warp level, but not the thread block level. There are implications regarding this difference such as no OpenACC `async`/`wait` in parallel regions or kernels.

In general, you might observe that OpenMP was built when threads were limited and start up overhead was considerable (as it still is on CPUs). The design reflects the need to control for this. OpenACC starts with devices built around thousands of very, very lightweight threads.

They are also complementary and can be used together very well.

OpenMP 4.0 Data Migration

The most obvious improvements for accelerators are the data migration commands. These look very similar to OpenACC.

```
#pragma omp target device(0) map(tofrom:B)
```

OpenMP vs. OpenACC Data Constructs

OpenMP

- target data
- target enter data
- target exit data
- target update
- declare target

OpenACC

- data
- enter data
- exit data
- update
- declare

OpenMP vs. OpenACC Data Clauses

OpenMP

OpenACC

- map(
- map(
- map(
- map(
- map(
- map(

OpenMP 5 has also embraced the NVIDIA "unified shared memory" paradigm

```
#pragma omp requires unified_shared_memory

complex_deep_data * cdp = create_array_of_data();

#pragma omp target    //Notice no mapping clauses!
operate_on_data( cdp );
```

Just like with NVIDIA Unified Memory, this is hopelessly naïve and is not used in production code. It is often recommended for a "first pass" (but I find that counter-productive).

The closely related deep copy directives (*declare mapper*) can be useful to aid in moving pointer-based data. As can the *allocate* clauses.

OpenMP vs. OpenACC Compute Constructs

OpenMP	OpenACC
target	parallel / kernels
teams	parallel / kernels
distribute	loop gang
parallel	parallel / kernels
for / do	loop worker or loop gang
simd	loop vector
is_device_ptr(...)	deviceptr(...)

OpenMP vs. OpenACC Differences

OpenMP	OpenACC
device(n)	---
depend(to:a)	async(n)
depend(from:b)	async(n)
nowait	async
loops, tasks, sections	loops
atomic	atomic
master, single, critical, barrier, locks, ordered, flush, cancel	---

SAXPY in OpenMP 4.0 on NVIDIA

```
int main(int argc, const char* argv[]) {
    int n = 10240; floata = 2.0f; floatb = 3.0f;
    float*x = (float*) malloc(n * sizeof(float));
    float*y = (float*) malloc(n * sizeof(float));

// Run SAXPY TWICE inside data region
#pragma omp target data map(to:x)
{
#pragma omp target map(tofrom:y)
#pragma omp teams
#pragma omp distribute
#pragma omp parallel for
    for(inti = 0; i < n; ++i){
          y[i] = a*x[i] + y[i];
     }
#pragma omp target map(tofrom:y)
#pragma omp teams
#pragma omp distribute
#pragma omp parallel for
     for(inti = 0; i < n; ++i){
          y[i] = b*x[i] + y[i];
     }
}
}
```

Courtesy Christian Terboven

Comparing OpenACC with OpenMP 4.0 on NVIDIA & Phi

OpenMP 4.0 for Intel Xeon Phi

```
#pragma omp target device(0) map(tofrom:B)
#pragma omp parallel for
for (i=0; i<N; i++)
    B[i] += sin(B[i]);
```

OpenMP 4.0 for NVIDIA GPU

```
#pragma omp target device(0) map(tofrom:B)
#pragma omp teams num_teams(num_blocks) num_threads(bsize)
#pragma omp distribute
for (i=0; i<N; i += num_blocks)
      #pragma omp parallel for
      for (b = i; b < i+num_blocks; b++)
            B[b] += sin(B[b]);
```

OpenACC for NVIDIA GPU

```
#pragma acc kernels
for (i=0; i<N; ++i)
     B[i] += sin(B[i]);
```

First two examples
Courtesy Christian Terboven

OpenMP 4.0 Across Architectures

OpenMP now has a number of OpenACC-like metadirectives to help cope with this confusion:

```
#pragma omp target map(to:a,b) map(from:c)
#pragma omp metadrictive when (device={arch(nvptx)}: teams loop) default (parallel loop)
for (i = 1; i<n; i++)
    c[i] = a[i] * b[i]
```

And also variant functions to substitute code for different targets.

```
#pragma omp declare target
int some_routine(int a){
   //do things in serial way
}

#pragma omp declare variant ( int some_routine(int a) ) match(context={target} \
                                                          device={arch(nvptx)} )
int some_routine_gpu(int a){
    //gpu optimized code
}
```

OpenMP 4.0 Across Compilers

Cray C Compiler (v8.5)

```
#pragma omp target teams distribute
for(int ii = 0; ii < y; ++ii)
```

Clang Compiler (alpha)

```
#pragma omp target teams distribute \
        parallel for chedule(static,1)
for(int ii = 0; ii < y; ++ii)
```

Intel C Compiler (v16.0)

```
#pragma omp target
#pragma omp parallel
for for(int ii = 0; ii < y; ++ii)
```

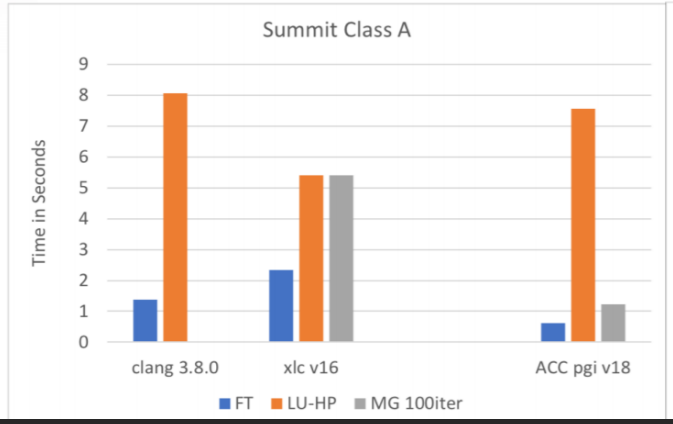
GCC C Compiler (v6.1))

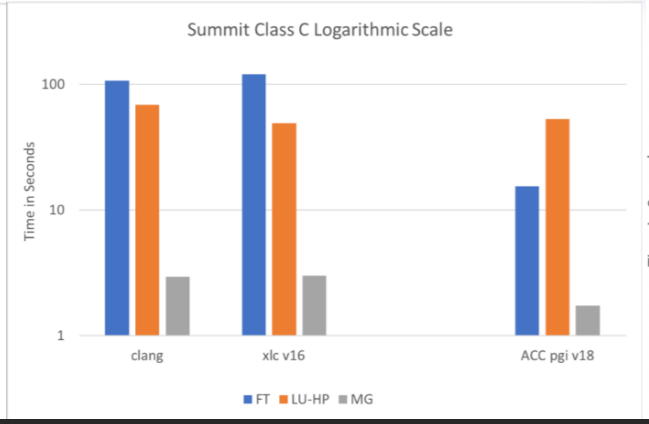
```
#pragma omp target teams distribute \
        parallel for
for(int ii = 0; ii < y; ++ii)
```

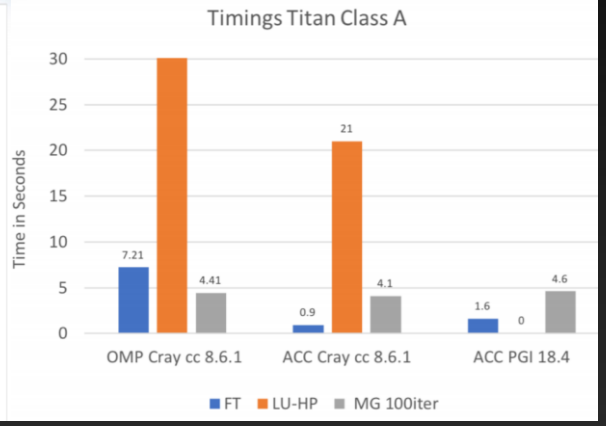
Subtle and confusing? You bet.
For a nice discussion of these examples visit their authors at
https://www.openmp.org/wp-content/uploads/Matt_openmp-booth-talk.pdf

Latest Data.

From the excellent paper *Is OpenMP 4.5 Target Off-load Ready for Real Life? A Case Study of Three Benchmark Kernels (Diaz, Jost, Chandrasekaran, Pino)* we have some recent data:







In summary, using NPB benchmarks (FFT, Gauss Seidel, Multi-Grid) on leadership class platforms (Titan and Summit) using multiple compilers (clang, gcc, PGI, Cray, IBM), OpenMP is not yet competitive with OpenACC on GPUs.

A very interesting side-note is that OpenACC *kernels* and *loop* performed the same.

So, at this time…

- If you are using Phi Knights Corner or Knights Landing, you are probably going to be using the Intel OpenMP 4+ release. Unless you use it in cache mode, and then this is moot (more later).

- If you are using NVIDIA GPUs, you are going to be using OpenACC.

Of course, there are other ways of programming both of these devices. You might treat Phi as MPI cores and use CUDA on NVIDIA , for example. But if the directive based approach is for you, then your path is clear. I don't attempt to discuss the many other types of accelerators here (AMD, DSPs, FPGAs, ARM), but these techniques apply there as well.

And as you should now suspect, even if it takes a while for these to merge as a standard, it is not a big jump for you to move between them.

The national labs have decided to deal with this by adding an additional layer of abstraction that will translate to the most usable lower level API with frameworks such as oneAPI (Includes DPC++ and extends SYCL) or Kokkos.

Hybrid Programming

John Urbanic
Parallel Computing Specialist
Pittsburgh Supercomputing Center

Assuming you know basic MPI

- This is a rare group that can discuss this topic meaningfully.
- I have mentioned MPI 3.0's "improvements" to its hybrid capabilities. These are primarily tying up loose ends and formally specifying that things work as you would expect, and as they largely do. Your MPI 1/2 knowledge will be more than sufficient here.

Hybrid OpenACC Programming (Fast & Wrong)

```
#pragma acc data copy(Temperature_last), create(Temperature)
while ( dt_global > MAX_TEMP_ERROR && iteration <= max_iterations ) {

    #pragma acc kernels
    for(i = 1; i <= ROWS; i++) {
        for(j = 1; j <= COLUMNS; j++) {
            Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                        Temperature_last[i][j+1] + Temperature_last[i][j-1]);
        }
    }

    if(my_PE_num != npes-1){
        MPI_Send(&Temperature[ROWS][1], COLUMNS, MPI_DOUBLE, my_PE_num+1, DOWN, MPI_COMM_WORLD);
    }

    if(my_PE_num != 0){
        MPI_Recv(&Temperature_last[0][1], COLUMNS, MPI_DOUBLE, my_PE_num-1, DOWN, MPI_COMM_WORLD, &status);
    }

    if(my_PE_num != 0){
        MPI_Send(&Temperature[1][1], COLUMNS, MPI_DOUBLE, my_PE_num-1, UP, MPI_COMM_WORLD);
    }

    if(my_PE_num != npes-1){
        MPI_Recv(&Temperature_last[ROWS+1][1], COLUMNS, MPI_DOUBLE, my_PE_num+1, UP, MPI_COMM_WORLD, &status);
    }

    dt = 0.0;

    #pragma acc kernels
    for(i = 1; i <= ROWS; i++){
        for(j = 1; j <= COLUMNS; j++){
            dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
            Temperature_last[i][j] = Temperature[i][j];
        }
    }

    MPI_Reduce(&dt, &dt_global, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
    MPI_Bcast(&dt_global, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    if((iteration % 100) == 0) {
        if (my_PE_num == npes-1){
            #pragma acc update host(Temperature)
            track_progress(iteration);
        }
    }

    iteration++;
}
```

MPI routines using host data

0.9s

Hybrid OpenACC Programming (Slow and Right)

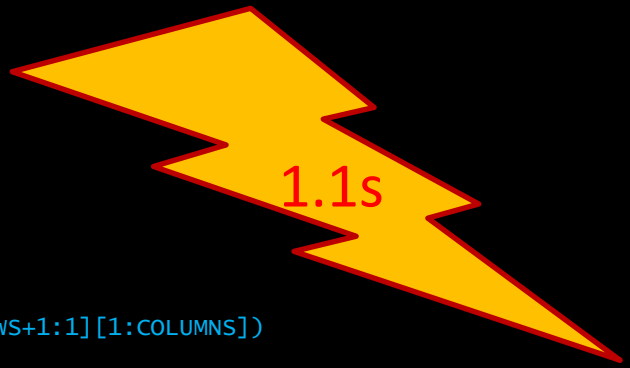
```
#pragma acc data copy(Temperature_last), create(Temperature)
while ( dt_global > MAX_TEMP_ERROR && iteration <= max_iterations ) {

    #pragma acc kernels
    for(i = 1; i <= ROWS; i++) {
        for(j = 1; j <= COLUMNS; j++) {
            Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                        Temperature_last[i][j+1] + Temperature_last[i][j-1]);
        }
    }

    #pragma acc update host(Temperature, Temperature_last)

    if(my_PE_num != npes-1){
        MPI_Send(&Temperature[ROWS][1], COLUMNS, MPI_DOUBLE, my_PE_num+1, DOWN, MPI_COMM_WORLD);
    }

    if(my_PE_num != 0){
        MPI_Recv(&Temperature_last[0][1], COLUMNS, MPI_DOUBLE, my_PE_num-1, DOWN, MPI_COMM_WORLD, &status);
    }

    if(my_PE_num != 0){
        MPI_Send(&Temperature[1][1], COLUMNS, MPI_DOUBLE, my_PE_num-1, UP, MPI_COMM_WORLD);
    }

    if(my_PE_num != npes-1){
        MPI_Recv(&Temperature_last[ROWS+1][1], COLUMNS, MPI_DOUBLE, my_PE_num+1, UP, MPI_COMM_WORLD, &status);
    }

    #pragma acc update device(Temperature, Temperature_last)

    dt = 0.0;

    #pragma acc kernels
    for(i = 1; i <= ROWS; i++){
        for(j = 1; j <= COLUMNS; j++){
            dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
            Temperature_last[i][j] = Temperature[i][j];
        }
    }

    MPI_Reduce(&dt, &dt_global, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
    MPI_Bcast(&dt_global, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    if((iteration % 100) == 0) {
        if (my_PE_num == npes-1){
            #pragma acc update host(Temperature)
            track_progress(iteration);
        }
    }

    iteration++;
}
```

Update data entering and leaving MPI section

9.3 s

Hybrid OpenACC Programming (Much Better)

```
#pragma acc data copy(Temperature_last), create(Temperature)
while ( dt_global > MAX_TEMP_ERROR && iteration <= max_iterations ) {

    #pragma acc kernels
    for(i = 1; i <= ROWS; i++) {
        for(j = 1; j <= COLUMNS; j++) {
            Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                        Temperature_last[i][j+1] + Temperature_last[i][j-1]);
        }
    }

    #pragma acc update host(Temperature[1:1][1:COLUMNS],Temperature[ROWS:1][1:COLUMNS])

    if(my_PE_num != npes-1){
        MPI_Send(&Temperature[ROWS][1], COLUMNS, MPI_DOUBLE, my_PE_num+1, DOWN, MPI_COMM_WORLD);
    }

    if(my_PE_num != 0){
        MPI_Recv(&Temperature_last[0][1], COLUMNS, MPI_DOUBLE, my_PE_num-1, DOWN, MPI_COMM_WORLD, &status);
    }

    if(my_PE_num != 0){
        MPI_Send(&Temperature[1][1], COLUMNS, MPI_DOUBLE, my_PE_num-1, UP, MPI_COMM_WORLD);
    }

    if(my_PE_num != npes-1){
        MPI_Recv(&Temperature_last[ROWS+1][1], COLUMNS, MPI_DOUBLE, my_PE_num+1, UP, MPI_COMM_WORLD, &status);
    }

    #pragma acc update device(Temperature_last[0:1][1:COLUMNS], Temperature_last[ROWS+1:1][1:COLUMNS])

    dt = 0.0;

    #pragma acc kernels
    for(i = 1; i <= ROWS; i++){
        for(j = 1; j <= COLUMNS; j++){
            dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
            Temperature_last[i][j] = Temperature[i][j];
        }
    }

    MPI_Reduce(&dt, &dt_global, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
    MPI_Bcast(&dt_global, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    if((iteration % 100) == 0) {
        if (my_PE_num == npes-1){
            #pragma acc update host(Temperature)
            track_progress(iteration);
        }
    }

    iteration++;
}
```



Mix and Match

- PGI Compile:

```
mpicc -acc laplace_hybrid.c
mpf90 -acc laplace_hybrid.f90
mpicc -mp -acc laplace_hybrid.c
etc…
```

- Running:

```
interact ?
     -n 4
     -N1 -n4
     -p GPU -N1 -n4
     -p GPU -N4 -n4
     -N1 -n28
     -N4 -n112
     etc…
```

- Intel bonus detail:

```
export I_MPI_PIN_DOMAIN=omp     (or you may not actually get multiple cores!)
Details at https://software.intel.com/en-us/articles/hybrid-applications-intelmpi-openmp
```

Bottom Line…

- Each one of these approaches occupies its own space.
- If you understand this, you will not be confused as to how they fit together.
- Once again…

In Conclusion…