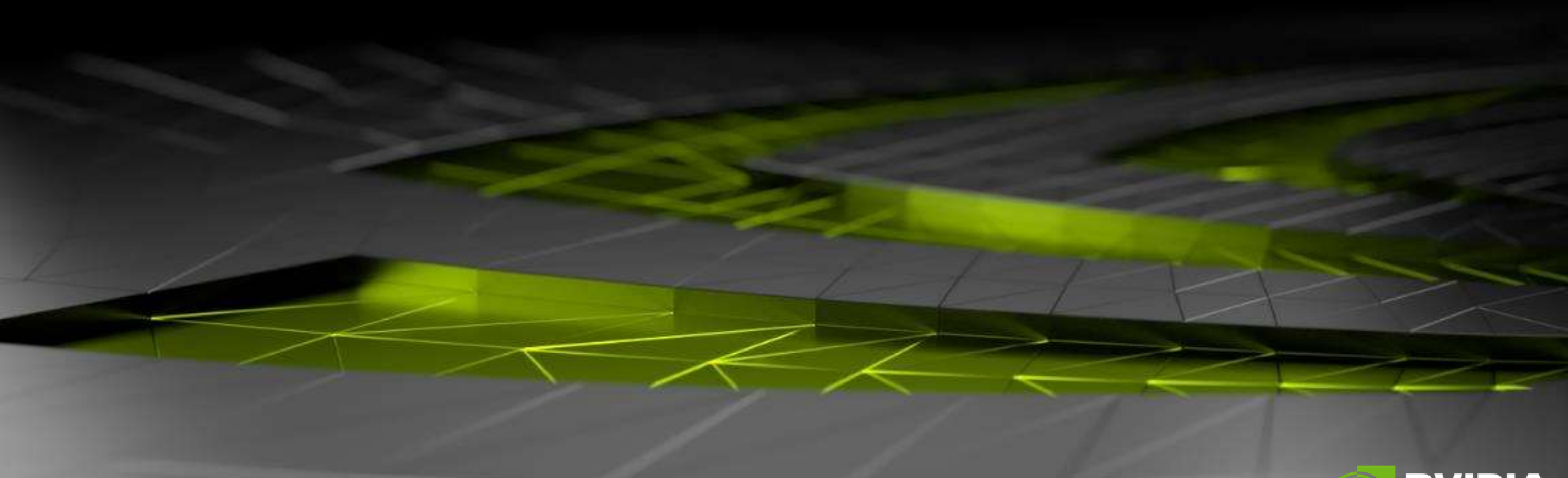
ECOLE IN2P3

Palaiseau, May 27th 2016

Programming heterogeneous architectures with directives OpenMP 4.5 and OpenACC 2.5



François Courteille |Senior Solutions Architect, NVIDIA |fcourteille@nvidia.com

Agenda

AGENDA

1 TESLA GPUs Roadmap

2 Tesla Platform for Users & Developers

3 History of OpenMP & OpenACC

4 Standards difference

5 Portability Challenges

6 Case Study : Jacobi iteration

7 Conclusions

AGENDA 1/2

History of OpenMP & OpenACC

Standards difference

- Philosophical Differences
- Technical Differences

Portability Challenges

Case Study : Jacobi iteration

Conclusions

AGENDA 2/2

Case Study : Jacobi

1/OpenMP step by step Case Study

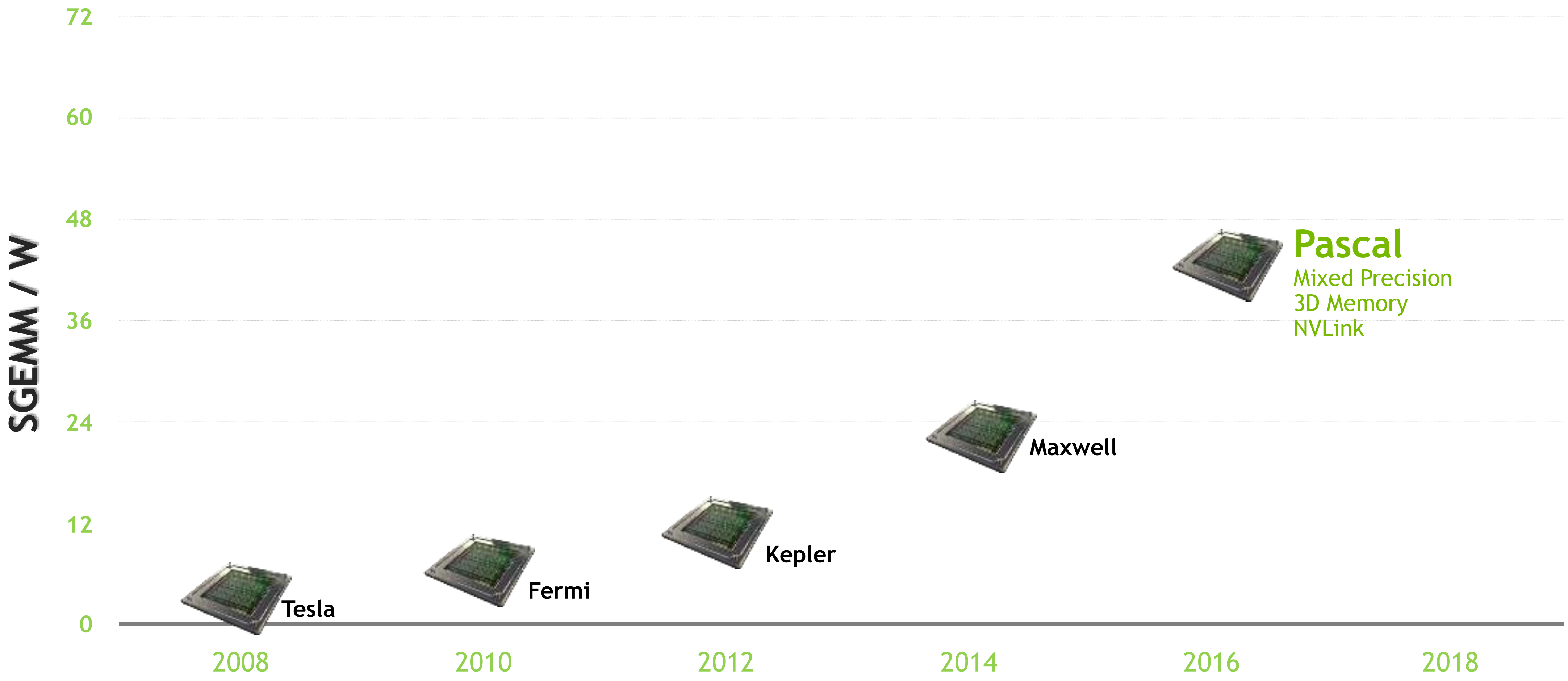
- Parallelize on CPU
- Offload to GPU
- Team Up
- Increase Parallelism
- Improve Scheduling

Additional Experiments

2/OpenACC step by step Case Study

- Parallelize on CPU/GPU
- Manage data locality
- Unified Memory

GPU ARCHITECTURE ROADMAP



END-TO-END PRODUCT FAMILY

HYPERSCALE HPC

Tesla M4, M40

Hyperscale deployment for DL training, inference, video & image processing

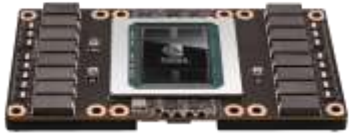
MIXED-APPS HPC

Tesla K80

HPC data centers running mix of CPU and GPU workloads

STRONG-SCALING HPC

Tesla P100



Hyperscale & HPC data centers running apps that scale to multiple GPUs

FULLY INTEGRATED DL SUPERCOMPUTER

DGX-1

For customers who need to get going now with fully integrated solution

NVIDIA DGX-1
WORLD’S FIRST DEEP LEARNING SUPERCOMPUTER

170 TFLOPS FP16

8x Tesla P100 16GB

NVLink Hybrid Cube Mesh

Accelerates Major AI Frameworks

Dual Xeon

7 TB SSD Deep Learning Cache

Dual 10GbE, Quad IB 100Gb

3RU – 3200W

TESLA PLATFORM FOR USERS & DEVELOPERS

COMMON PROGRAMMING MODELS ACROSS MULTIPLE CPUS

Libraries

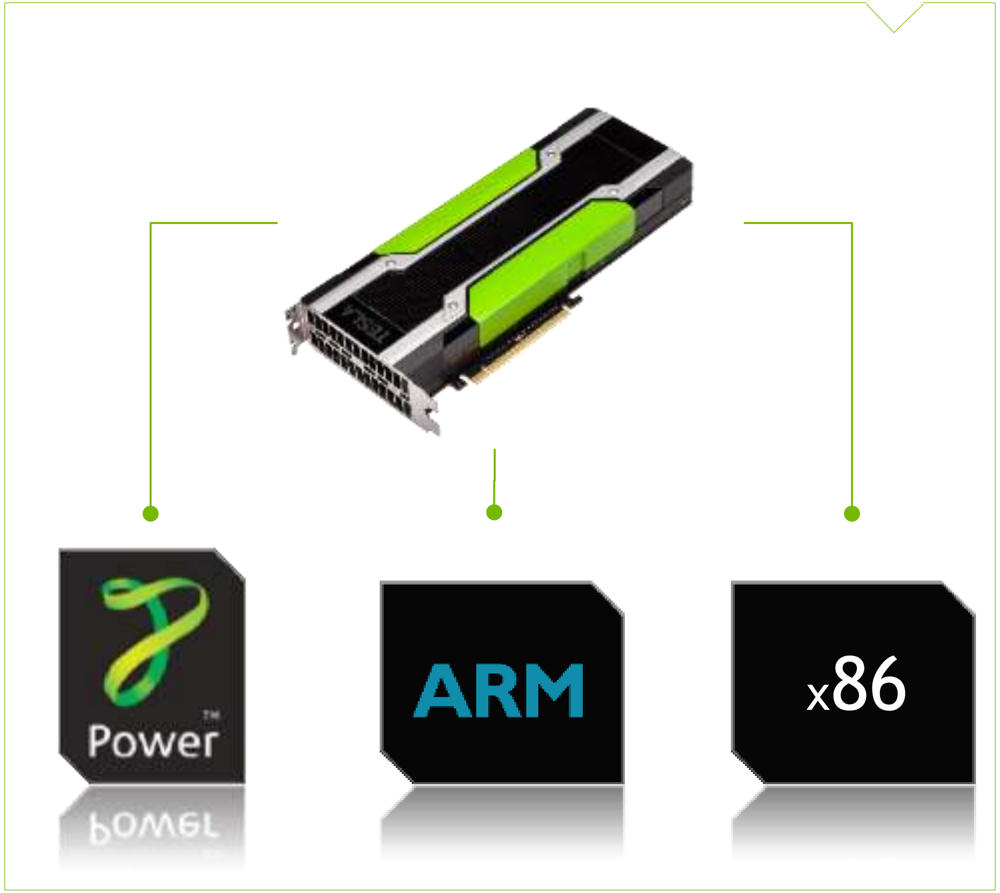
AmgX cuDNN cuBLAS OpenCV Thrust

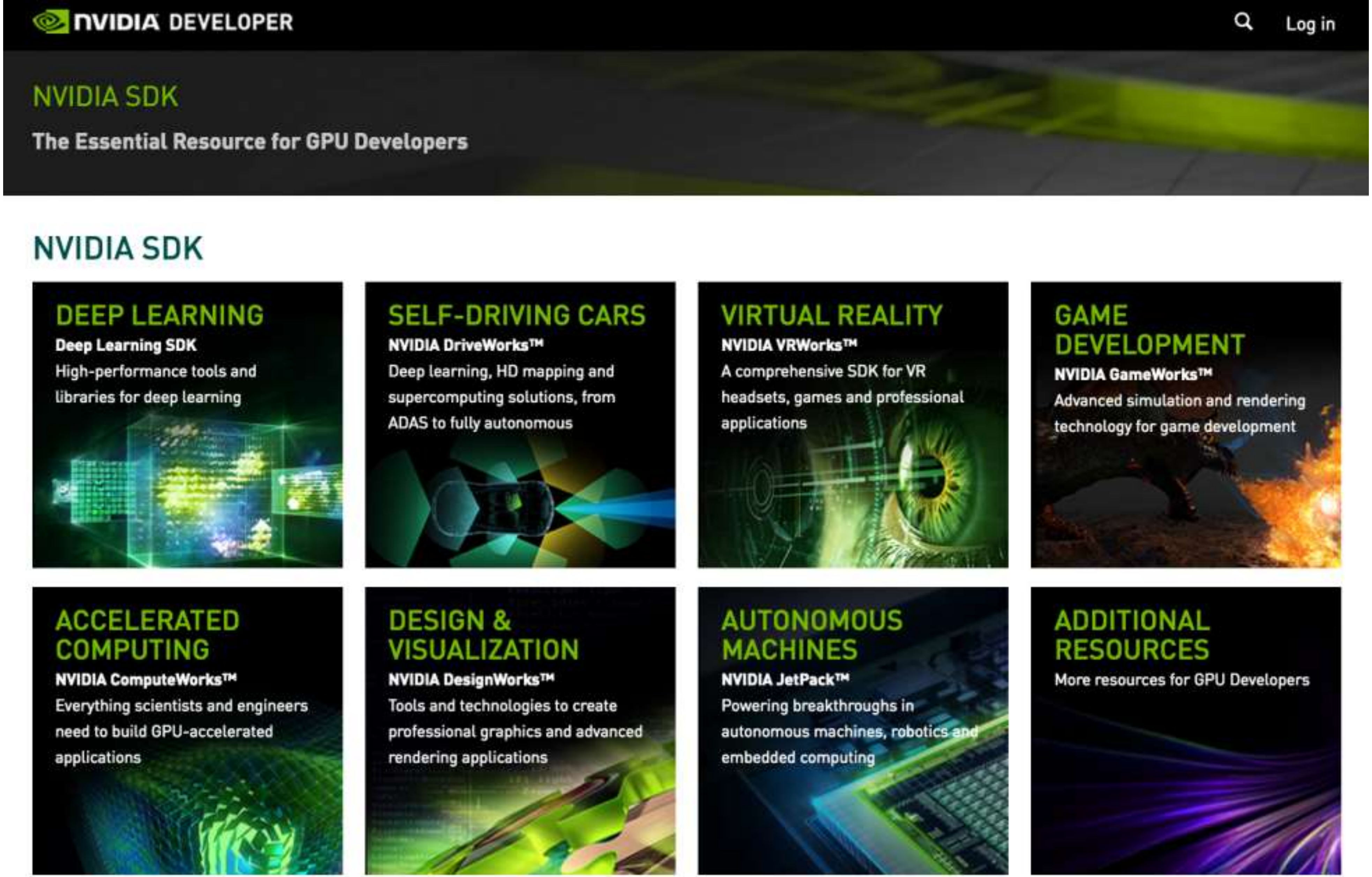
Compiler Directives

OpenACC

Programming Languages

C/C++ Fortran python Java





NVIDIA SDK

The Essential Resource for GPU Developers

NVIDIA SDK

DEEP LEARNING

Deep Learning SDK

High-performance tools and libraries for deep learning

SELF-DRIVING CARS

NVIDIA DriveWorks™

Deep learning, HD mapping and supercomputing solutions, from ADAS to fully autonomous

VIRTUAL REALITY

NVIDIA VRWorks™

A comprehensive SDK for VR headsets, games and professional applications

GAME DEVELOPMENT

NVIDIA GameWorks™

Advanced simulation and rendering technology for game development

ACCELERATED COMPUTING

NVIDIA ComputeWorks™

Everything scientists and engineers need to build GPU-accelerated applications

DESIGN & VISUALIZATION

NVIDIA DesignWorks™

Tools and technologies to create professional graphics and advanced rendering applications

AUTONOMOUS MACHINES

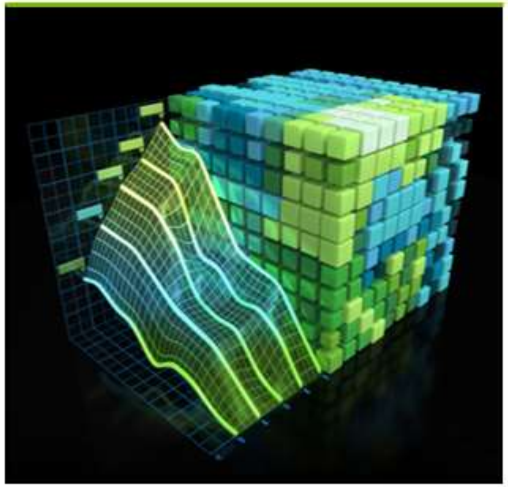
NVIDIA JetPack™

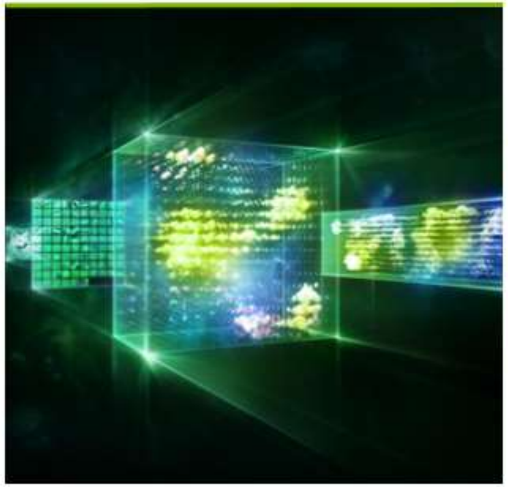
Powering breakthroughs in autonomous machines, robotics and embedded computing

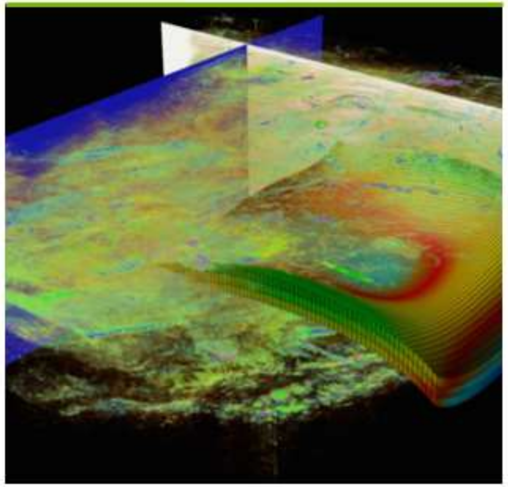
ADDITIONAL RESOURCES

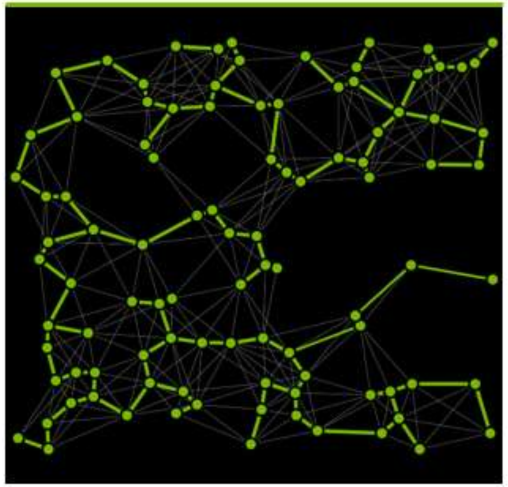
More resources for GPU Developers

NVIDIA SDK: COMPUTEWORKS


CUDA


cuDNN


IndeX


nvGRAPH

And other technologies such as:
AMGx, cuSOLVER, cuSPARSE, OpenACC, NSIGHT, THRUST

INTRODUCING THE NEW OPENACC TOOLKIT

Free Toolkit Offers Simple & Powerful Path to Accelerated Computing

OpenACC

http://developer.nvidia.com/openacc

PGI Compiler
Free OpenACC compiler for academia

PGProf Profiler
Easily find where to add compiler directives

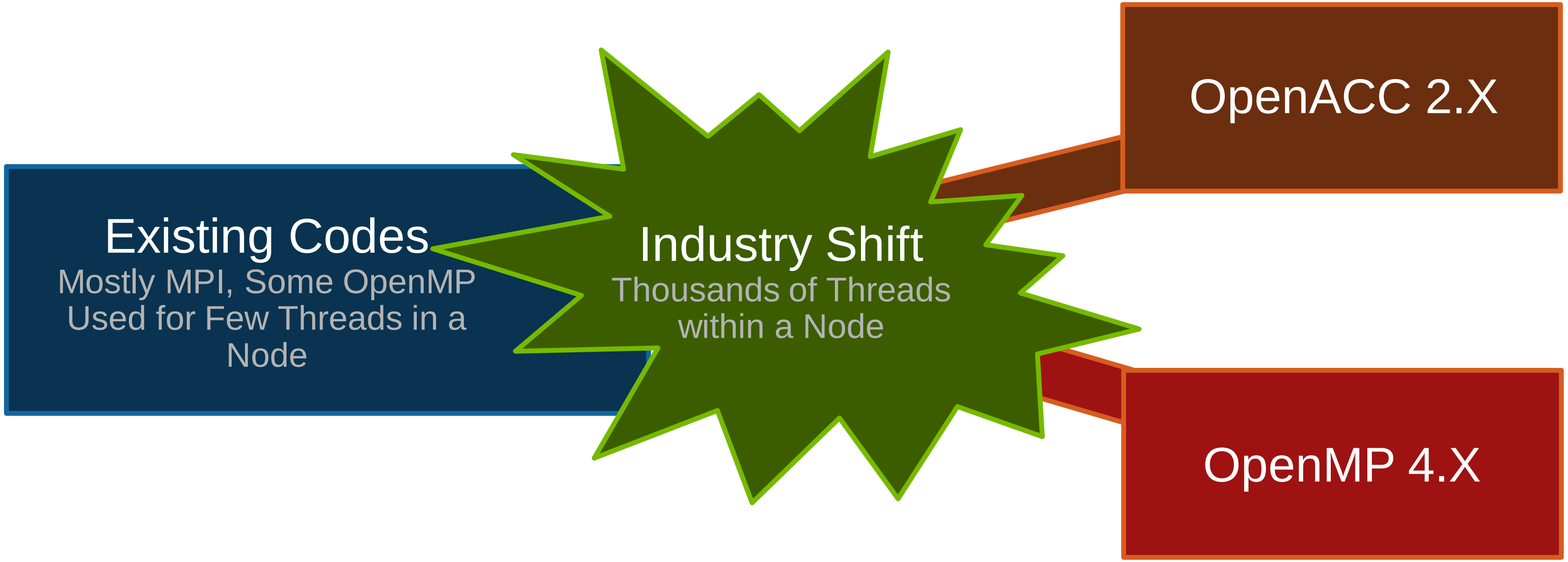
GPU Wizard
Identify which GPU libraries can jumpstart code

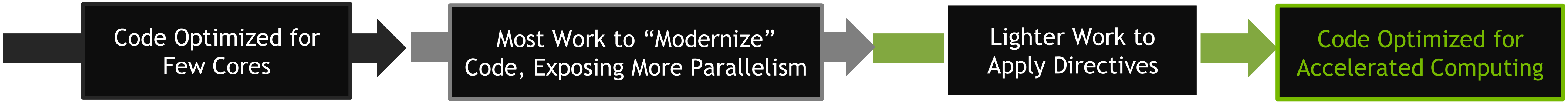
Code Samples
Learn from examples of real-world algorithms

Documentation
Quick start guide, Best practices, Forums

COMPARING OPENACC 2.5 AND OPENMP 4.5

EXISTING HPC CODES MUST ADAPT TO ACCELERATED COMPUTING


Existing Codes
Mostly MPI, Some OpenMP Used for Few Threads in a Node
Industry Shift
Thousands of Threads within a Node
OpenACC 2.X
OpenMP 4.X


Code Optimized for Few Cores
Most Work to “Modernize” Code, Exposing More Parallelism
Lighter Work to Apply Directives
Code Optimized for Accelerated Computing

A TALE OF TWO SPECS.

A BRIEF HISTORY OF OPENMP

1996 - Architecture Review Board (ARB) formed by several vendors implementing their own directives for Shared Memory Parallelism (SMP).

1997 - 1.0 was released for C/C++ and Fortran with support for parallelizing loops across threads.

2000, 2002 – Version 2.0 of Fortran, C/C++ specifications released.

2005 - Version 2.5 released, combining both specs into one.

2008 - Version 3.0 released, added support for tasking

2011 - Version 3.1 release, improved support for tasking

2013 - Version 4.0 released, added support for offloading (and more)

2015 - Version 4.5 released, improved support for offloading targets (and more)

A BRIEF HISTORY OF OPENACC

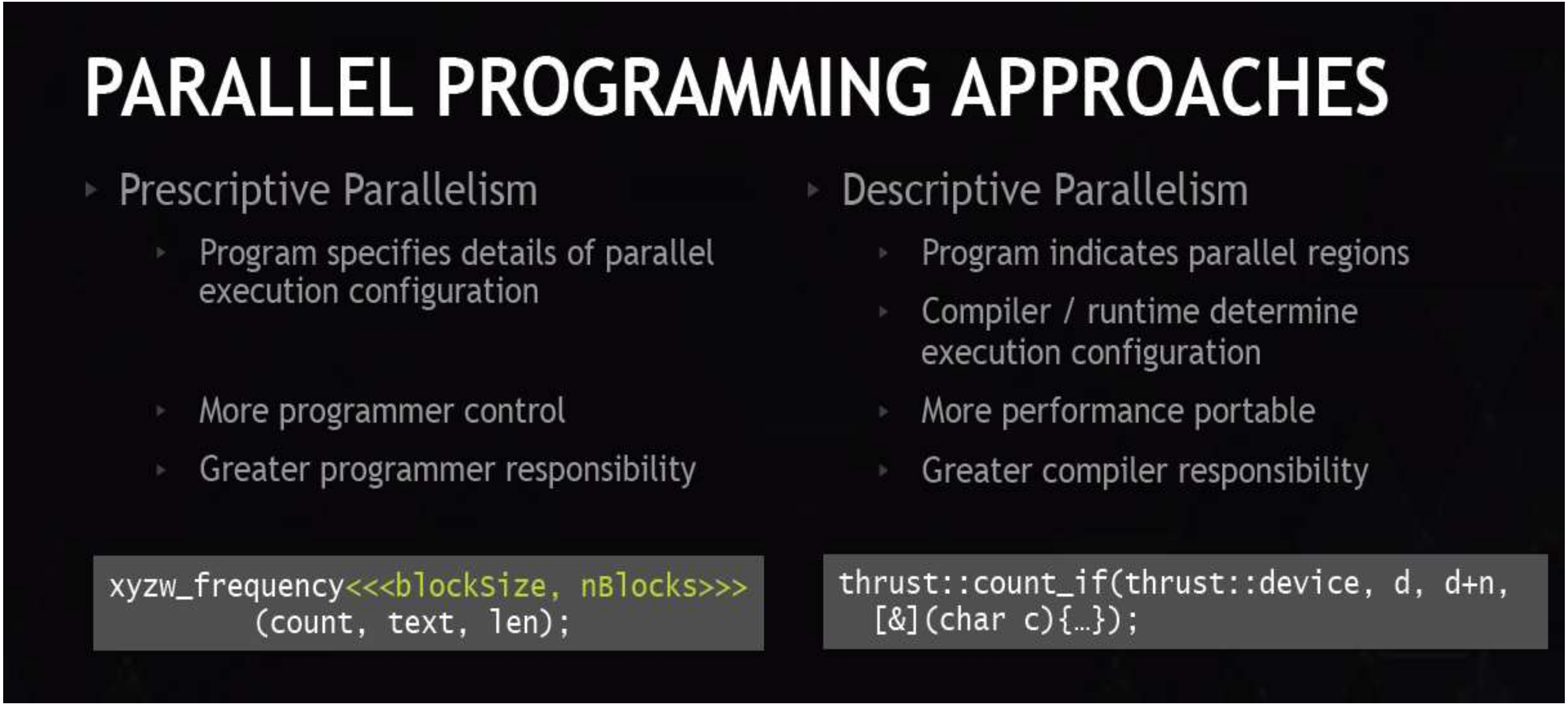
2010 - OpenACC founded by CAPS, Cray, PGI, and NVIDIA, to unify directives for accelerators being developed by CAPS, Cray, and PGI independently

2011 - OpenACC 1.0 released

2013 - OpenACC 2.0 released, adding support for unstructured data management and clarifying specification language

2015 - OpenACC 2.5 released, contains primarily clarifications with some additional features.

PHILOSOPHICAL DIFFERENCES



PARALLEL PROGRAMMING APPROACHES

- Prescriptive Parallelism
  - Program specifies details of parallel execution configuration
  - More programmer control
  - Greater programmer responsibility

```
xyzw_frequency<<<blockSize, nBlocks>>>
        (count, text, len);
```

- Descriptive Parallelism
  - Program indicates parallel regions
  - Compiler / runtime determine execution configuration
  - More performance portable
  - Greater compiler responsibility

```
thrust::count_if(thrust::device, d, d+n,
  [&](char c){…});
```

http://on-demand.gputechconf.com/gtc/2015/presentation/S5820-Mark-Harris.pdf

OPENMP: COMPILERS ARE DUMB, USERS ARE SMART. RESTRUCTURING NON-PARALLEL CODE IS OPTIONAL.

OPENACC: COMPILERS CAN BE SMART AND SMARTER WITH THE USER’S HELP. NON-PARALLEL CODE MUST BE MADE PARALLEL.

PHILOSOPHICAL DIFFERENCES

OpenMP:

The OpenMP API covers only user-directed parallelization, wherein *the programmer explicitly specifies* the actions to be taken by the compiler and runtime system in order to execute the program in parallel.

The OpenMP API *does not cover* compiler-generated automatic parallelization and directives to the compiler to assist such parallelization.

OpenACC:

The programming model allows the programmer to *augment information available to the compilers*, including specification of data local to an accelerator, *guidance on mapping of loops* onto an accelerator, and similar performance-related details.

PHILOSOPHICAL TRADE-OFFS

OpenMP

- Consistent, predictable behavior between implementations
- Users can parallelize non-parallel code and protect data races explicitly
- Some optimizations are off the table
- Substantially different architectures require substantially different

OpenACC

- Quality of implementation will greatly affect performance
- Users must restructure their code to be parallel and free of data races
- Compiler has more freedom and information to optimize
- High level parallel directives can be applied to different architectures by the compiler

TECHNICAL DIFFERENCES

OPENACC & OPENMP: KEY DIFFERENCE

OpenACC Goal

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

  !$acc kernels

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

end subroutine saxpy
```

Simple, Descriptive Code

Single Code for CPUs, GPUs, Phis

OpenMP Goal

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

  !$omp target teams
  !$omp& distribute parallel do
  do i=1,n
     y(i) = a*x(i)+y(i)
  enddo
  !$omp end target teams &
  !$omp& distribute parallel do

end subroutine saxpy
```

Many Ways to Write Same Code

```
!$omp target teams distribute &
!$omp& parallel do simd
```

```
!$omp parallel do num_threads
(284) &
!$omp& simd safelen(16)
```

Explicit, Parallel Code

Different Codes Optimized for CPUs, GPUs, or Phis

SAXPY - SINGLE PRECISION A*X PLUS Y

SAXPY in C

```c
void saxpy(int n, float a,
           float *x, float *y)
{

  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

int N = 1<<20;



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

SAXPY in Fortran

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

  do i=1,n
    y(i) = a*x(i)+y(i)
  enddo

end subroutine saxpy


...
! Perform SAXPY on N elements
call saxpy(N, 2.0, x, y)
...
```

SAXPY - SINGLE PRECISION A*X PLUS Y IN OPENMP - CPU

SAXPY in C

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

int N = 1<<20;


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

SAXPY in Fortran

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

...
! Perform SAXPY on N elements
call saxpy(N, 2.0, x, y)
...
```

SAXPY - SINGLE PRECISION A*X PLUS Y IN OPENACC - CPU & ACCELERATOR

SAXPY in C

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

int N = 1<<20;

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

SAXPY in Fortran

```fortran
subroutine saxpy(n, a, x, y)
  real :: x(*), y(*), a
  integer :: n, i
!$acc parallel loop
  do i=1,n
    y(i) = a*x(i)+y(i)
  enddo
!$acc end parallel
end subroutine saxpy
...
! Perform SAXPY on N elements
call saxpy(N, 2.0, x, y)
...
```

SAXPY - SINGLE PRECISION A*X PLUS Y IN OPENMP - ACCELERATOR (GPU)

SAXPY in C

```
void saxpy(int n, float a,
        float *x, float *y)
{
# #pragma omp target teams \
        distribute parallel for
for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

int N = 1<<20;

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

SAXPY in Fortran

```
subroutine saxpy(n, a, x, y)
  real :: x(*), y(*), a
  integer :: n, i
!$omp  target teams &
  !$omp& distribute parallel do
  do i=1,n
    y(i) = a*x(i)+y(i)
  enddo
!$omp end target teams &
  !$omp& distribute parallel do
end subroutine saxpy
...
! Perform SAXPY on N elements
call saxpy(N, 2.0, x, y)
...
```

PARALLEL: SIMILAR, BUT DIFFERENT

- OMP Parallel

- Creates a *team* of *threads*
- Very well-defined how the number of threads is chosen.
- May synchronize within the team
- Data races are the user’s responsibility

- ACC Parallel

- Creates 1 or more *gangs* of *workers*
- Compiler free to choose number of gangs, workers, vector length
- May not synchronize between gangs
- Data races not allowed

OMP TEAMS VS. ACC PARALLEL

- OMP Teams

- Creates a *league* of 1 or more *thread teams*
- Compiler free to choose number of teams, threads, and simd lanes.
- May not synchronize between teams
- Only available within target regions

- ACC Parallel

- Creates 1 or more *gangs* of *workers*
- Compiler free to choose number of gangs, workers, vector length
- May not synchronize between gangs
- May be used anywhere

COMPILER-DRIVEN MODE

- OpenMP

- Fully user-driven (no analogue)
- Some compilers choose to go above and beyond after applying OpenMP, but not guaranteed

- OpenACC

- `Kernels` directive declares desire to parallelize a region of code, but places the burden of analysis on the compiler
- Compiler required to be able to do analysis and make decisions.

LOOP: SIMILAR BUT DIFFERENT

- OMP Loop (For/Do)

- Splits (“*Workshares*”) the iterations of the next loop to threads in the team, guarantees the user has managed any data races

- Loop will be run over threads and scheduling of loop iterations may restrict the compiler

- ACC Loop

- Declares the loop iterations as independent & race free (parallel) or interesting & should be analyzed (kernels)

- User able to declare independence w/o declaring scheduling

- Compiler free to schedule with gangs/workers/vector, unless

DISTRIBUTE VS. LOOP

- OMP Distribute

- Must live in a TEAMS region
- Distributes loop iterations over 1 or more thread teams
- Only master thread of each team runs iterations, until PARALLEL is encountered
- Loop iterations are implicitly independent, but some compiler optimizations still restricted

- ACC Loop

- Declares the loop iterations as independent & race free (parallel) or interesting & should be analyzed (kernels)
- Compiler free to schedule with gangs/workers/vector, unless overridden by user

DISTRIBUTE EXAMPLE

```
#pragma omp target teams
{
#pragma omp distribute
  for(i=0; i<n; i++)
    for(j=0;j<m;j++)
      for(k=0;k<p;k++)
}
```

```
#pragma acc parallel
{
#pragma acc loop
  for(i=0; i<n; i++)
#pragma acc loop
    for(j=0;j<m;j++)
#pragma acc loop
      for(k=0;k<p;k++)
}
```

DISTRIBUTE EXAMPLE

```
#pragma omp target teams
{
#pragma omp distribute
  for(i=0; i<n; i++)
    for(j=0;j<m;j++)
      for(k=0;k<p;k++)
}
```

Generate a 1 or more thread teams

Distribute “i” over teams.

No information about “j” or “k” loops

```
#pragma acc parallel

#pragma acc loop
        i=0; i<n; i++)
          a acc loop
        r(j=0;j<m;j++)
#pragma acc loop
        for(k=0;k<p;k++)
}
```

DISTRIBUTE EXAMPLE

```
#pragma omp target teams
{
#pragma omp distr
   for(i=0; i<n; i
      for(j=0;j<m;j
         for(k=0;k<p;k++)
}
```

Generate a 1 or more gangs

These loops are independent, do the *right thing*

```
#pragma acc parallel
{
#pragma acc loop
   for(i=0; i<n; i++)
#pragma acc loop
      for(j=0;j<m;j++)
#pragma acc loop
         for(k=0;k<p;k++)
}
```

DISTRIBUTE EXAMPLE

```
#pragma omp target teams
{
#pragma omp distribute
  for(i=0; i<n; i++)
    for(j=0;j<m;j++)
      for(k=0;k<p;k++)
}
```

What’s the *right thing?*

Interchange? Distribute? Workshare?
Vectorize? Stripmine? Ignore? …

```
#pragma acc parallel
{
#pragma acc loop
  for(i=0; i<n; i++)
#pragma acc loop
    for(j=0;j<m;j++)
#pragma acc loop
      for(k=0;k<p;k++)
}
```

SYNCHRONIZATION

- OpenMP

- Users may use barriers, critical regions, and/or locks to protect data races
- It's possible to parallelize non-parallel code

- OpenACC

- Users expected to refactor code to remove data races.
- Code should be made truly parallel and scalable

SYNCHRONIZATION EXAMPLE

```
#pragma omp parallel private(p)
{
  funcA(p);
#pragma omp barrier
  funcB(p);
}
```

```
function funcA(p[N]){
  #pragma acc parallel
}
function funcB(p[N]){
  #pragma acc parallel
}
```

SYNCHRONIZATION EXAMPLE

```
#pragma omp parallel for
for (i=0; i<N; i++)
{
#pragma omp critical
  A[i] = rand();
  A[i] *= 2;
}
```

```
parallelRand(A);
#pragma acc parallel loop
for (i=0; i<N; i++)
{
  A[i] *= 2;
}
```

PORTABILITY CHALLENGES

How to Write Portable Code (OMP)

```
#ifdef GPU
#pragma omp target omp teams distribute parallel for reduction(max:error) \
            collapse(2) schedule(static,1)
#elif defined(CPU)
#pragma omp parallel for reduction(max:error)
#elif defined(SOMETHING_ELSE)
#pragma omp …
endif
        for( int j = 1; j < n-1; j++)
        {
#if defined(CPU) && defined(USE_SIMD)
#pragma omp simd
#endif
            for( int i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }
```

Ifdefs can be used to choose particular directives per device at compile-time

How to Write Portable Code (OMP)

```
#pragma omp \
#ifdef GPU
target teams distribute \
#endif
parallel for reduction(max:error) \
#ifdef GPU
collapse(2) schedule(static,1)
endif
        for( int j = 1; j < n-1; j++)
        {
            for( int i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }
```

Creative ifdefs might clean up the code, but still one target at a time.

How to Write Portable Code (OMP)

```
usegpu = 1;
#pragma omp target teams distribute parallel for reduction(max:error) \
#ifdef GPU
collapse(2) schedule(static,1) \
endif
if(target:usegpu)
        for( int j = 1; j < n-1; j++)
        {
            for( int i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }
```

The OpenMP if clause can help some too (4.5 improves this).

Note: This example assumes that a compiler will choose to generate 1 team when not in a target, making it the same as a standard “parallel for.”

How to Write Portable Code (ACC)

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

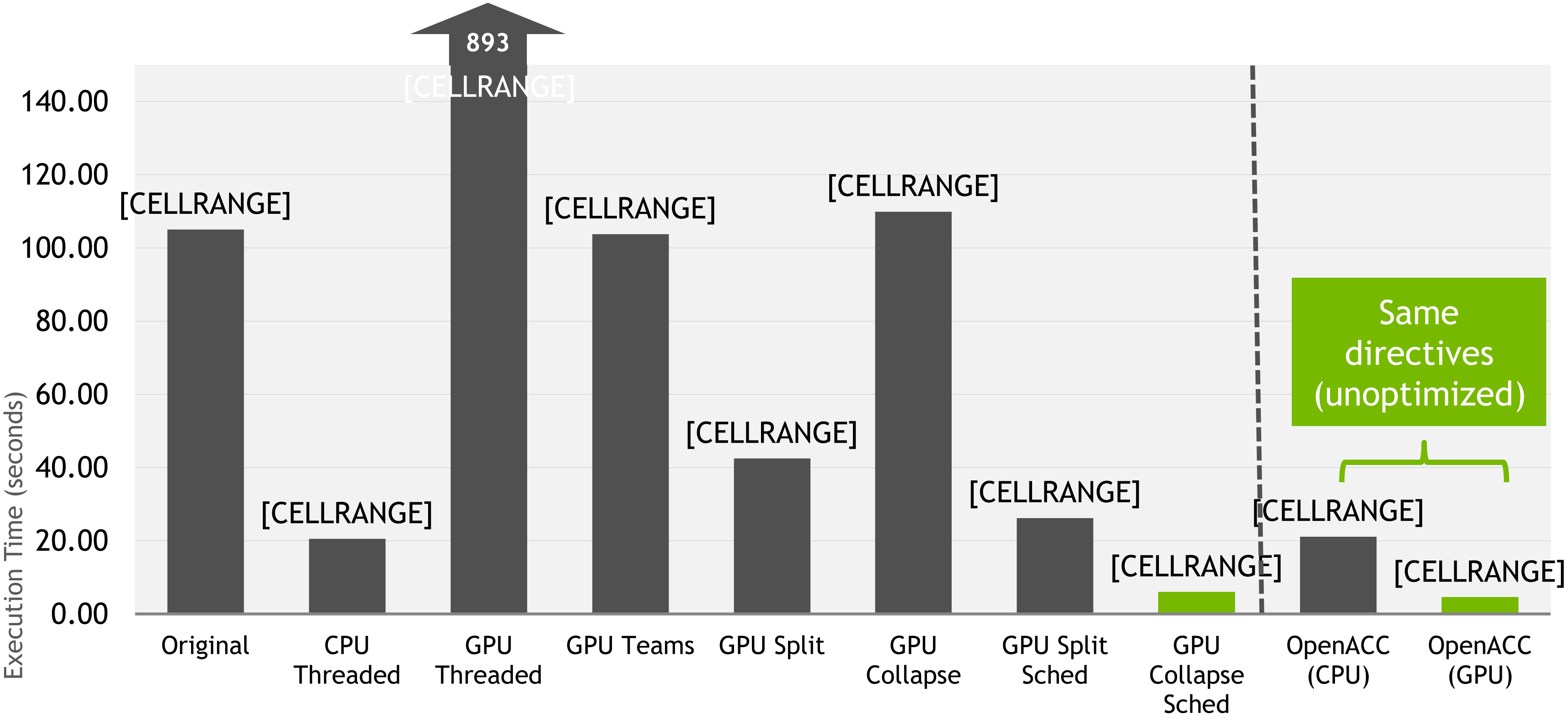
Developer presents the desire to parallelize to the compiler, compiler handles the rest.

How to Write Portable Code (ACC)

```
#pragma acc parallel loop reduction(max:error)
{
        for( int j = 1; j < n-1; j++)
        {
#pragma acc loop reduction(max:error)
            for( int i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }
}
```

Developer asserts the parallelism of the loops to the compiler, compiler makes decision about scheduling.

Execution Time (Smaller is Better)



NVIDIA Tesla K40, Intel Xeon E5-2690 v2 @ 3.00GHz - See GTC16 S6510 for additional information

COMPILER PORTABILITY (CPU)

- OpenMP

- Numerous well-tested implementations

- PGI, IBM, Intel, GCC, Cray, …

- OpenACC

- CPU implementations beginning to emerge

- X86: PGI

- ARM: PathScale

- Power: Coming soon

COMPILER PORTABILITY (OFFLOAD)

- OpenMP
- Few mature implementations
  - Intel (Phi)
  - Cray (GPU, *Phi?*)
  - GCC (Phi, GPUs in development)
  - Clang (Multiple targets in development)

- OpenACC
- Multiple mature implementations
  - PGI (NVIDIA & AMD)
  - PathScale (NVIDIA & AMD)
  - Cray (NVIDIA)
  - GCC (in development – starting with GCC 6.1)

TOOLCHAINS

OPENACC WITH PGI TOOLCHAIN

Optimize Once, Run Everywhere with OpenACC

		ARM CPU
	OpenPOWER CPU	OpenPOWER CPU
	x86 Xeon Phi	x86 Xeon Phi
x86 CPU	x86 CPU	x86 CPU
AMD GPU	AMD GPU	AMD GPU
NVIDIA GPU	NVIDIA GPU	NVIDIA GPU
2015	**2016**	**2017**

PGI Roadmaps are subject to change without notice.

OPENMP IN CLANG

Multi-vendor effort to implement OpenMP in Clang (including offloading)

Current status- interesting

How to get it- https://www.ibm.com/developerworks/community/blogs/8e0d7b52-b996-424b-bb33-345205594e0d?lang=en

OPENMP IN CLANG

How to get it, our way

Step one - make sure you have: gcc, cmake, python and cuda installed and updated

Step two - Look at

http://llvm.org/docs/GettingStarted.html

https://www.ibm.com/developerworks/community/blogs/8e0d7b52-b996-424b-bb33-345205594e0d?lang=en

Step three -

```
git clone https://github.com/clang-ykt/llvm_trunk.git
cd llvm_trunk/tools
git clone https://github.com/clang-ykt/clang_trunk.git clang
cd ../projects
git clone https://github.com/clang-ykt/openmp.git
```

OPENMP IN CLANG

How to build it

```
cd ..
mkdir build
cd build
cmake  -DCMAKE_BUILD_TYPE=DEBUG|RELEASE|MinSizeRel \
-DLLVM_TARGETS_TO_BUILD=“X86;NVPTX” \
   -DCMAKE_INSTALL_PREFIX=“<where you want it>" \
   -DLLVM_ENABLE_ASSERTIONS=ON \
   -DLLVM_ENABLE_BACKTRACES=ON \
   -DLLVM_ENABLE_WERROR=OFF \
   -DBUILD_SHARED_LIBS=OFF \
   -DLLVM_ENABLE_RTTI=ON \
   -DCMAKE_C_COMPILER=“GCC you want used" \
   -DCMAKE_CXX_COMPILER=“G++ you want used" \
   -G "Unix Makefiles" \  !there are other options, I like this one
   ../llvm_trunk
make [-j#]
make install
```

OPENMP IN CLANG

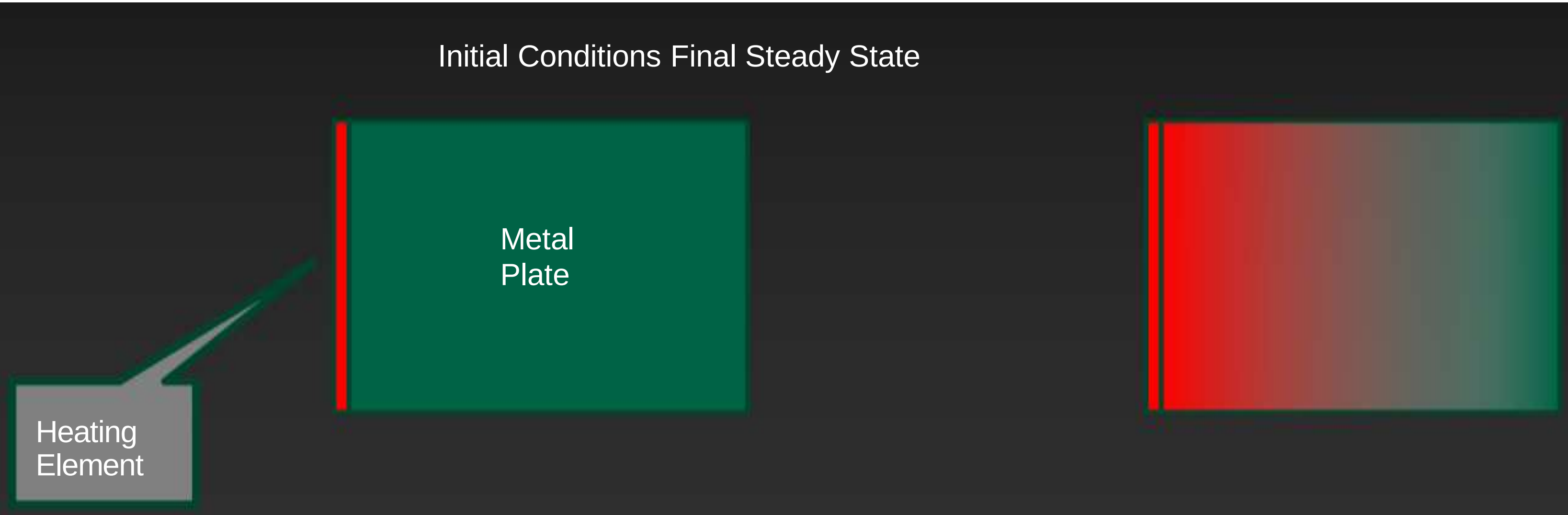
How to use it

```
export LIBOMP_LIB=<llvm-install-lib>
export OMPTARGET_LIBS=$LIBOMP_LIB
export LIBRARY_PATH=$OMPTARGET_LIBS
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$OMPTARGET_LIBS
export PATH=$PATH:<llvm_install-bin>
clang -O3 -fopenmp=libomp -omptargets=nvptx64sm_35-nvidia-linux …
```

Case Study: Jacobi Iteration

Our Foundation Exercise: Jacobi Iteration

- It is a simulation problem, not rigged for OpenACC.
- In this most basic form, it solves the Laplace equation: $\Delta u = 0$
- In our workshop example it is the Steady State Heat Equation.
- Students start with a realistic, normal serial code and parallelize it themselves



SINGLE EXAMPLE ABOUT HOW TO EXPRESS PARALLELISM AND DATA LOCALITY USING COMPILER DIRECTIVES LANGUAGES USING A GPU ACCELERATOR

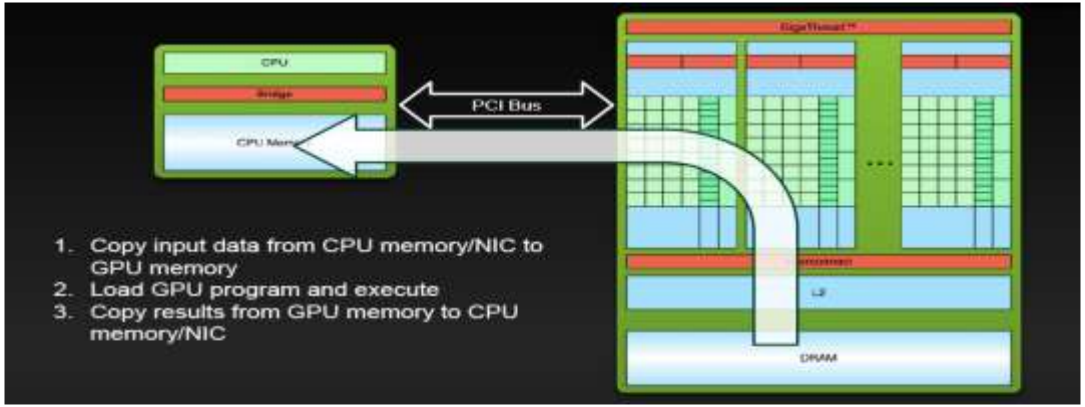
Identify Parallelism

Express Parallelism

Express Data Locality

Optimize

Data must be transfered between CPU and GPU memories



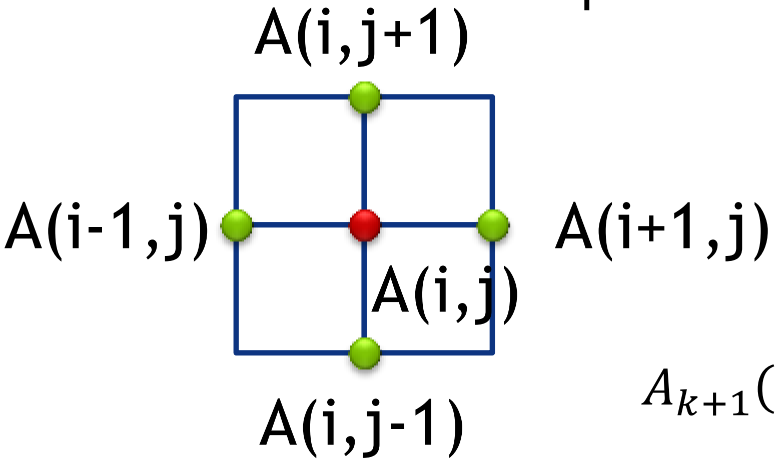
1. Copy input data from CPU memory/NIC to GPU memory
2. Load GPU program and execute
3. Copy results from GPU memory to CPU memory/NIC

EXAMPLE: JACOBI ITERATION

Iteratively converges to correct value (e.g. Temperature), by computing new values at each point from the average of neighboring points.

Common, useful algorithm

Example: Solve Laplace equation in 2D: $\nabla^2 \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y}) = \mathbf{0}$



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

JACOBI ITERATION

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


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

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

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


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

  iter++;
}
```

← Convergence Loop

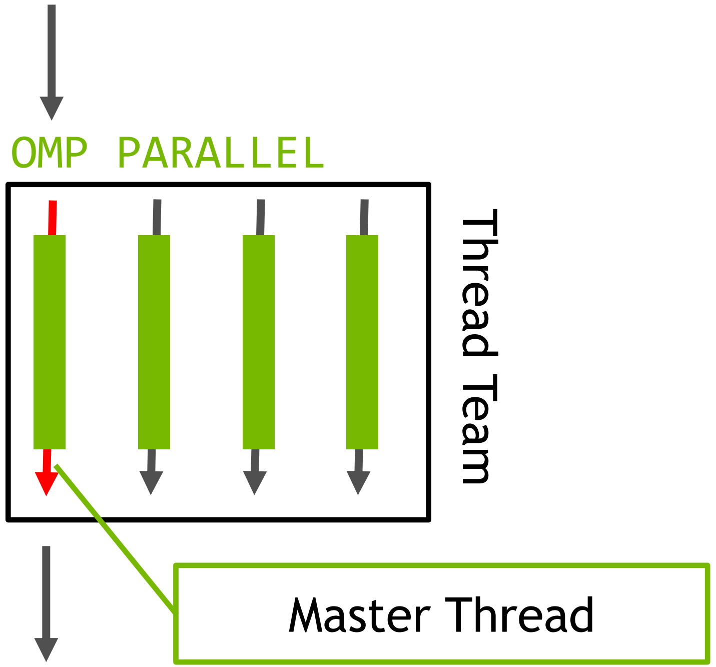
← Calculate Next

← Exchange Values

Parallelize on the CPU

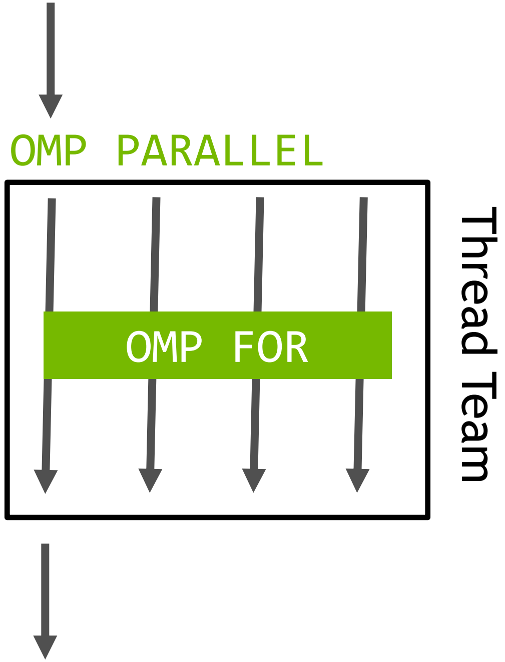
OPENMP WORKSHARING

- PARALLEL Directive
- Spawns a *team* of *threads*
- Execution continues redundantly on all threads of the team.
- All threads join at the end and the *master* thread continues execution.



OPENMP WORKSHARING

- FOR/DO (Loop) Directive
- Divides ("*workshares*") the iterations of the next loop across the threads in the team
- How the iterations are divided is determined by a *schedule*.



CPU-PARALLELISM

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

#pragma omp parallel for reduction(max:error)
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                + A[j-1][i] + A[j+1][i]);
            error = fmax( error, fabs(Anew[j][i] - A[j][i]));
        }
    }

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

    if(iter++ % 100 == 0) printf("%5d, %0.6f\n", iter, error);
}
```

Create a team of threads and workshare this loop across those threads.

Create a team of threads and workshare this loop across those threads.

CPU-PARALLELISM

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

#pragma omp parallel
{
#pragma omp for reduction(max:error)
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                + A[j-1][i] + A[j+1][i]);
            error = fmax( error, fabs(Anew[j][i] - A[j][i]));
        }
    }
#pragma omp barrier
#pragma omp for
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
}
    if(iter++ % 100 == 0) printf("%5d, %0.6f\n", iter, error);
}
```

← Create a team of threads

← Workshare this loop

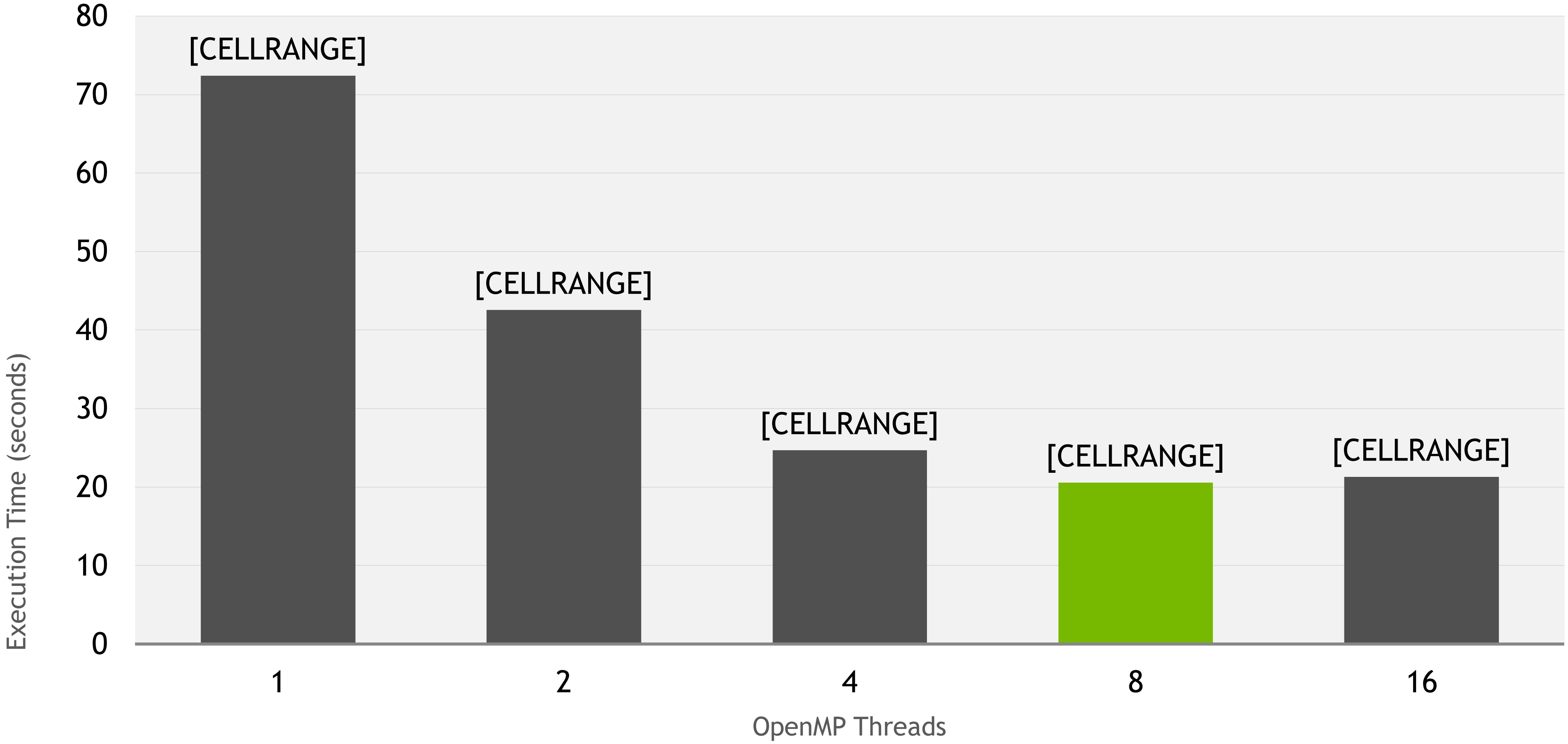
← Prevent threads from executing the second loop nest until the first completes

CPU-PARALLELISM

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

#pragma omp parallel for reduction(max:error)
    for( int j = 1; j < n-1; j++) {
#pragma omp simd
        for( int i = 1; i < m-1; i++ ) {
            Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                + A[j-1][i] + A[j+1][i]);
            error = fmax( error, fabs(Anew[j][i] - A[j][i]));
        }
    }

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

    if(iter++ % 100 == 0) printf("%5d, %0.6f\n", iter, error);
}
```

Some compilers want a SIMD directive to *simdize* on CPUS.

CPU Scaling (Smaller is Better)



Intel Xeon E5-2690 v2 @ 3.00GHz

Targeting the GPU

OPENMP OFFLOADING

TARGET Directive

Offloads execution and associated data from the CPU to the GPU

- The *target device* owns the data, accesses by the CPU during the execution of the target region are forbidden.
- Data used within the region may be implicitly or explicitly *mapped* to the device.
- All of OpenMP is allowed within target regions, but only a subset will run well on GPUs.

TARGET THE GPU

```
while ( error > tol && iter < iter_max )
{
    error = 0.0;
#pragma omp target
{
#pragma omp parallel for reduction(max:error)
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                + A[j-1][i] + A[j+1][i]);
            error = fmax( error, fabs(Anew[j][i] - A[j][i]));
        }
    }

#pragma omp parallel for
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
}
    if(iter++ % 100 == 0) printf("%5d, %0.6f\n", iter, error);
}
```

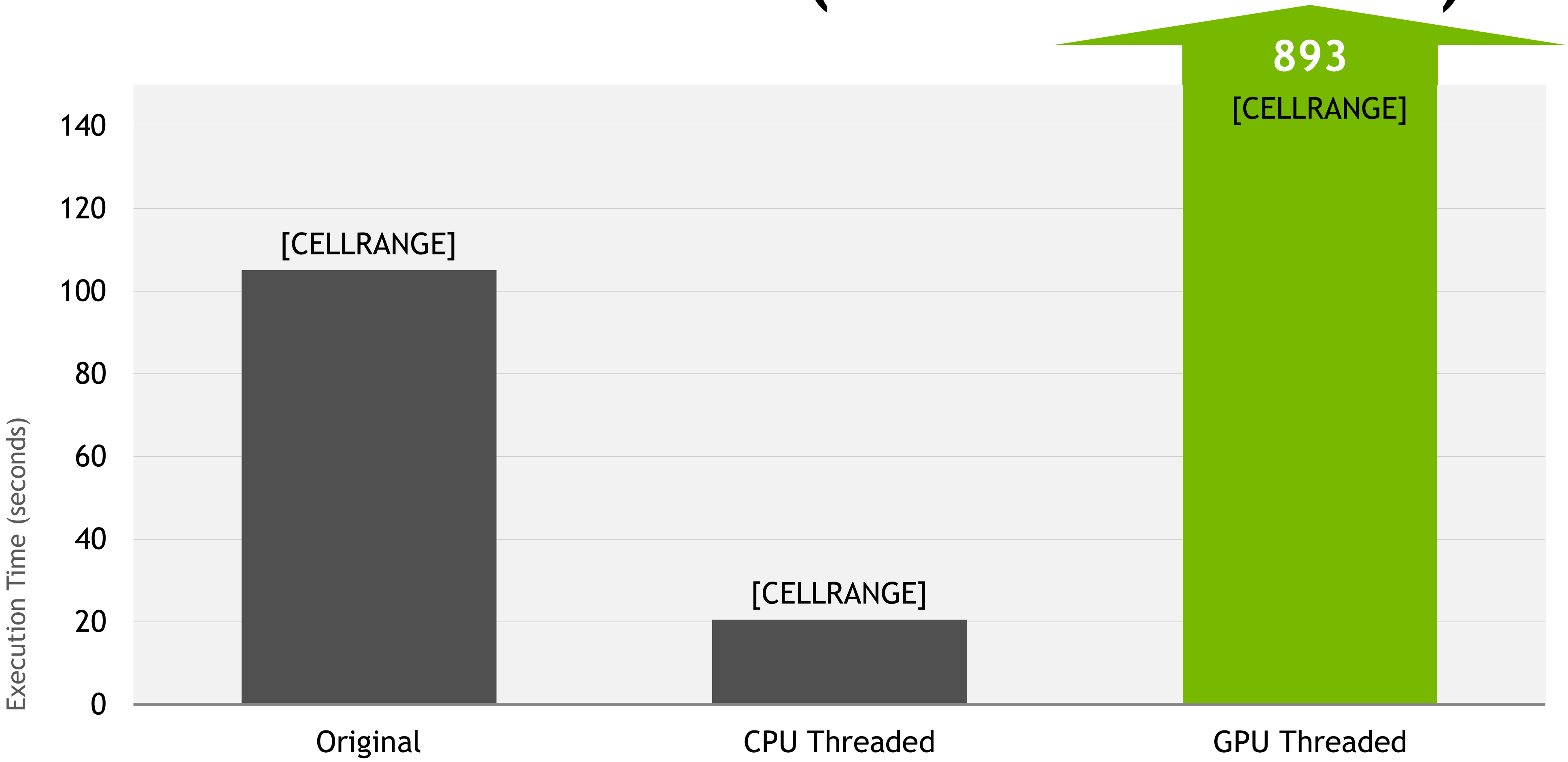
← Moves this region of code to the GPU and implicitly maps data.

TARGET THE GPU

```
while ( error > tol && iter < iter_max )
{
    error = 0.0;
#pragma omp target map(alloc:Anew[:n+2][:m+2]) map(tofrom:A[:n+2][:m+2])
{
#pragma omp parallel for reduction(max:error)
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                + A[j-1][i] + A[j+1][i]);
            error = fmax( error, fabs(Anew[j][i] - A[j][i]));
        }
    }

#pragma omp parallel for
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
}
    if(iter++ % 100 == 0) printf("%5d, %0.6f\n", iter, error);
}
```

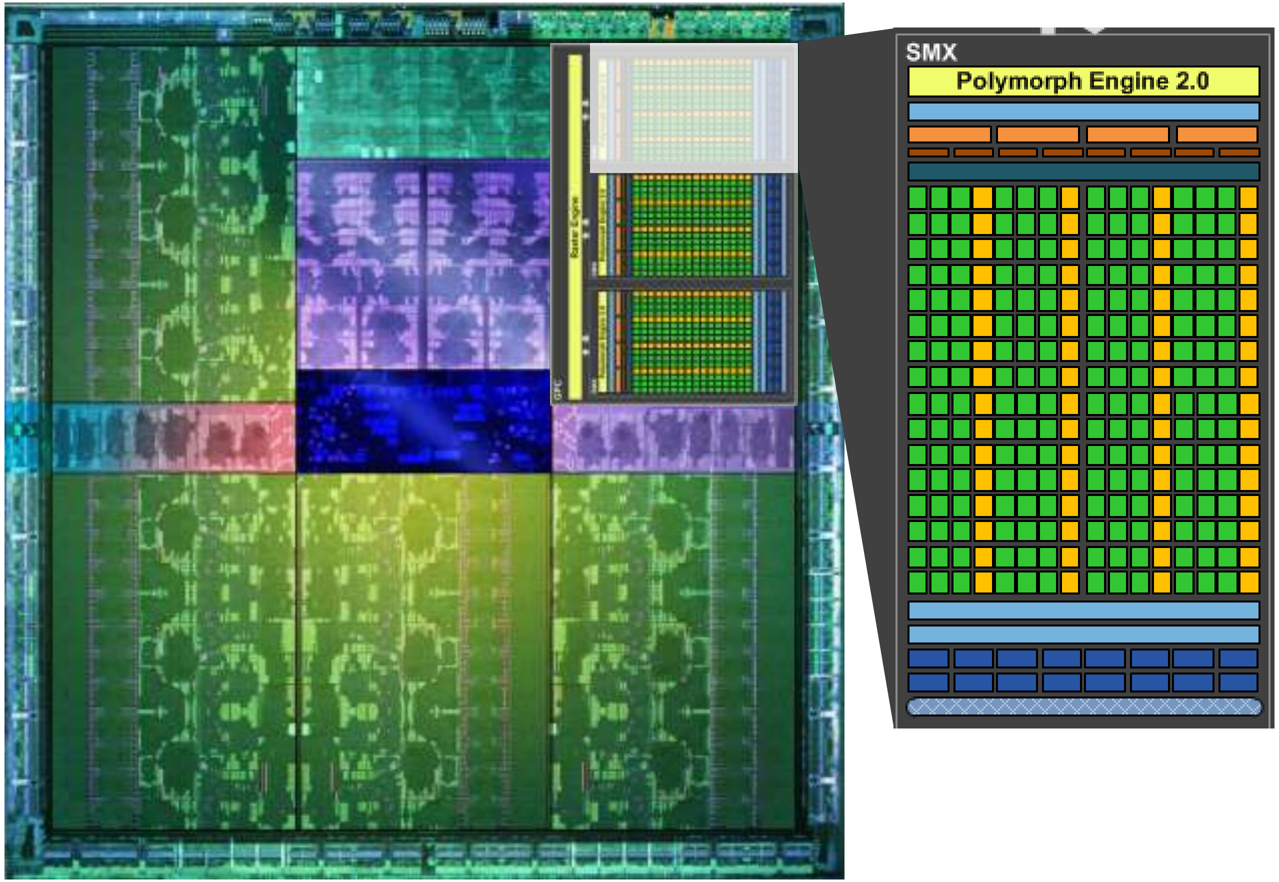
Moves this region of code to the GPU and explicitly maps data.

Execution Time (Smaller is Better)



NVIDIA Tesla K40, Intel Xeon E5-2690 v2 @ 3.00GHz

GPU ARCHITECTURE BASICS



GPUs are composed of 1 or more independent parts, known as *Streaming Multiprocessors* (*“SMs”*)

Threads are organized into *threadblocks*.

Threads within the same theadblock run on an SM and can synchronize.

Threads in different threadblocks (even if they’re on the same SM) cannot synchronize.

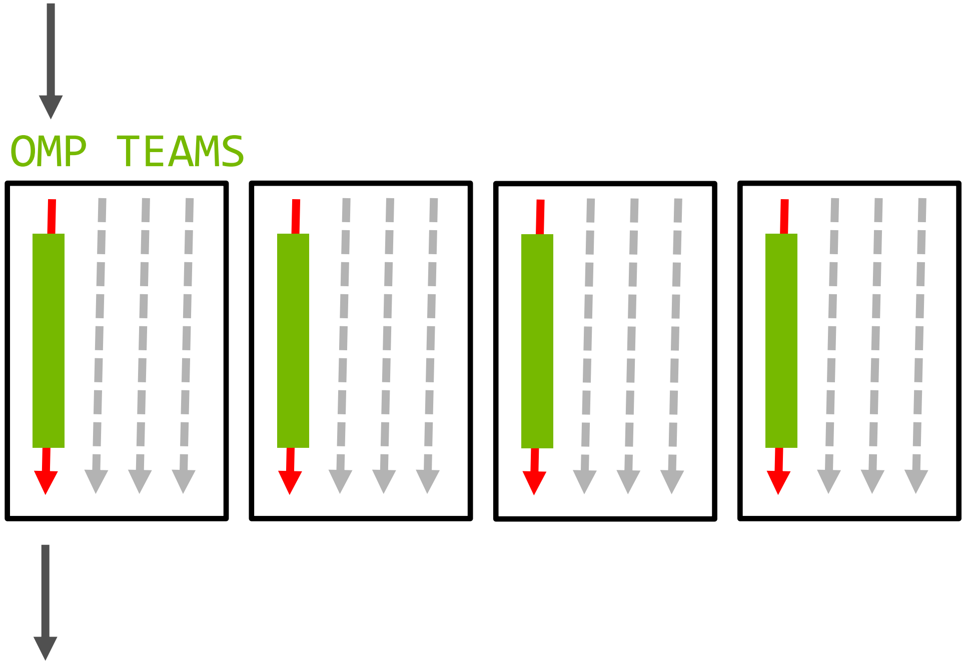
Teaming Up

OPENMP TEAMS

- TEAMS Directive

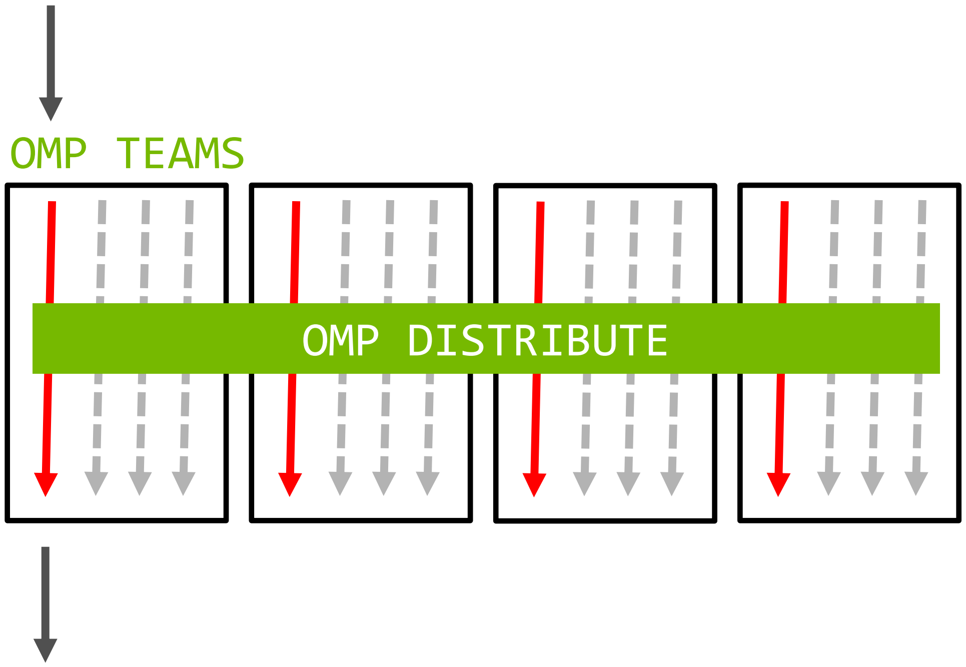
- To better utilize the GPU resources, use many thread teams via the TEAMS directive.

- Spawns 1 or more thread teams with the same number of threads
- Execution continues on the master threads of each team (redundantly)
- No synchronization between teams



OPENMP TEAMS

- DISTRIBUTE Directive
- Distributes the iterations of the next loop to the master threads of the teams.
  - Iterations are distributed statically.
  - There’s no guarantees about the order teams will execute.
  - No guarantee that all teams will execute simultaneously
  - Does not generate parallelism/worksharing within the thread teams



OPENMP DATA OFFLOADING

TARGET DATA Directive

Offloads data from the CPU to the GPU, but not execution

- The *target device* owns the data, accesses by the CPU during the execution of contained target regions are forbidden.
- Useful for sharing data between TARGET regions
- NOTE: A TARGET region *is a* TARGET DATA region.

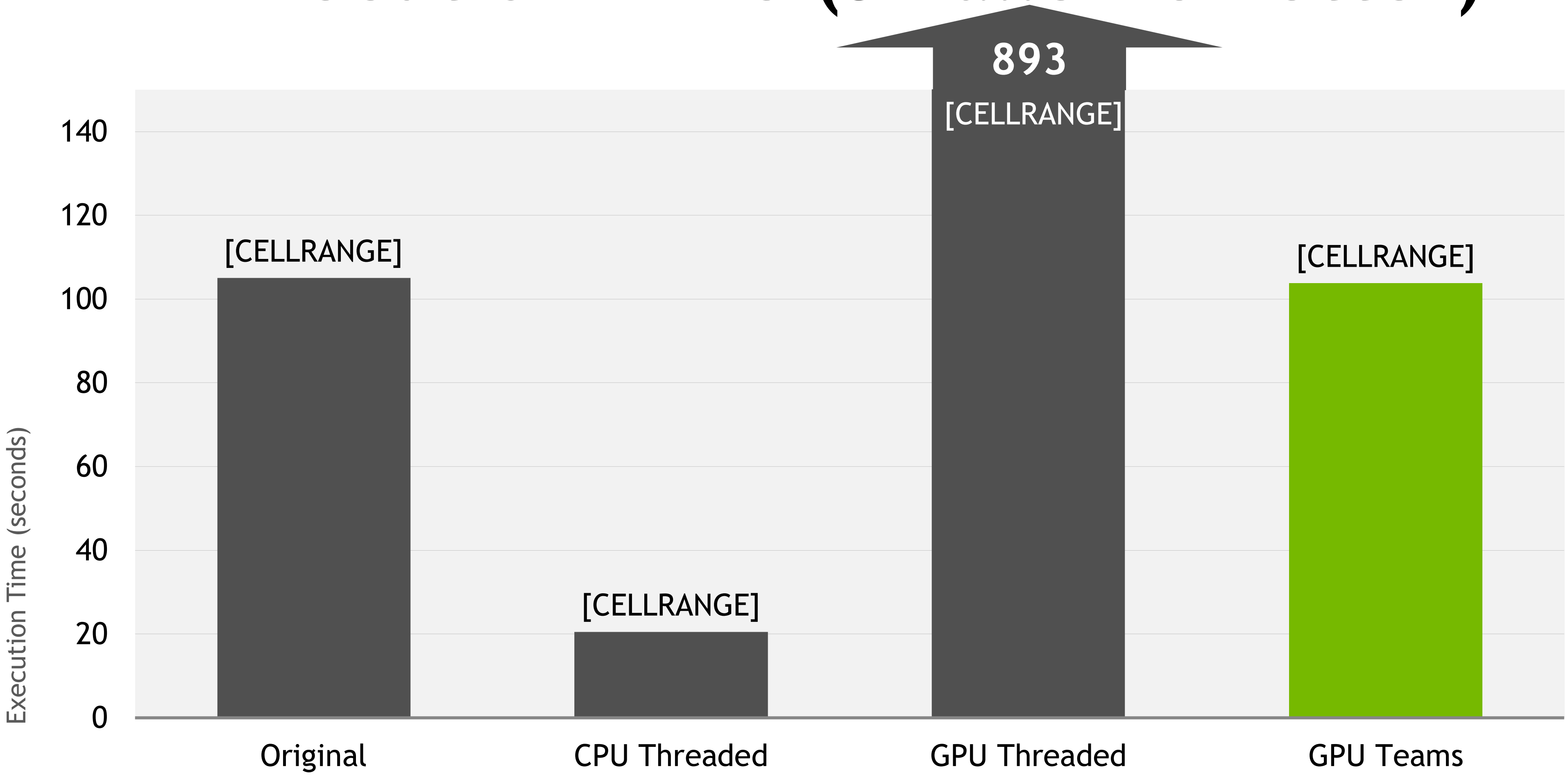
TEAMING UP

```
#pragma omp target data map(alloc:Anew) map(A)
    while ( error > tol && iter < iter_max )
    {
        error = 0.0;

#pragma omp target teams distribute parallel for reduction(max:error)
        for( int j = 1; j < n-1; j++)
        {
            for( int i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }

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

        if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);

        iter++;
    }
```

Explicitly maps arrays for the entire while loop.

- Spawns thread teams
- Distributes iterations to those teams
- Workshares within those teams.

Execution Time (Smaller is Better)



NVIDIA Tesla K40, Intel Xeon E5-2690 v2 @ 3.00GHz

Increasing Parallelism

INCREASING PARALLELISM

Currently both our distributed and workshared parallelism comes from the same loop.

- We could move the PARALLEL to the inner loop
- We could collapse them together

The COLLAPSE(N) clause

- Turns the next N loops into one, linearized loop.
- This will give us more parallelism to distribute, if we so choose.

Splitting Teams & Parallel

```
#pragma omp target teams distribute
        for( int j = 1; j < n-1; j++)
        {
#pragma omp parallel for reduction(max:error)
            for( int i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }

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

Distribute the “j” loop over teams.

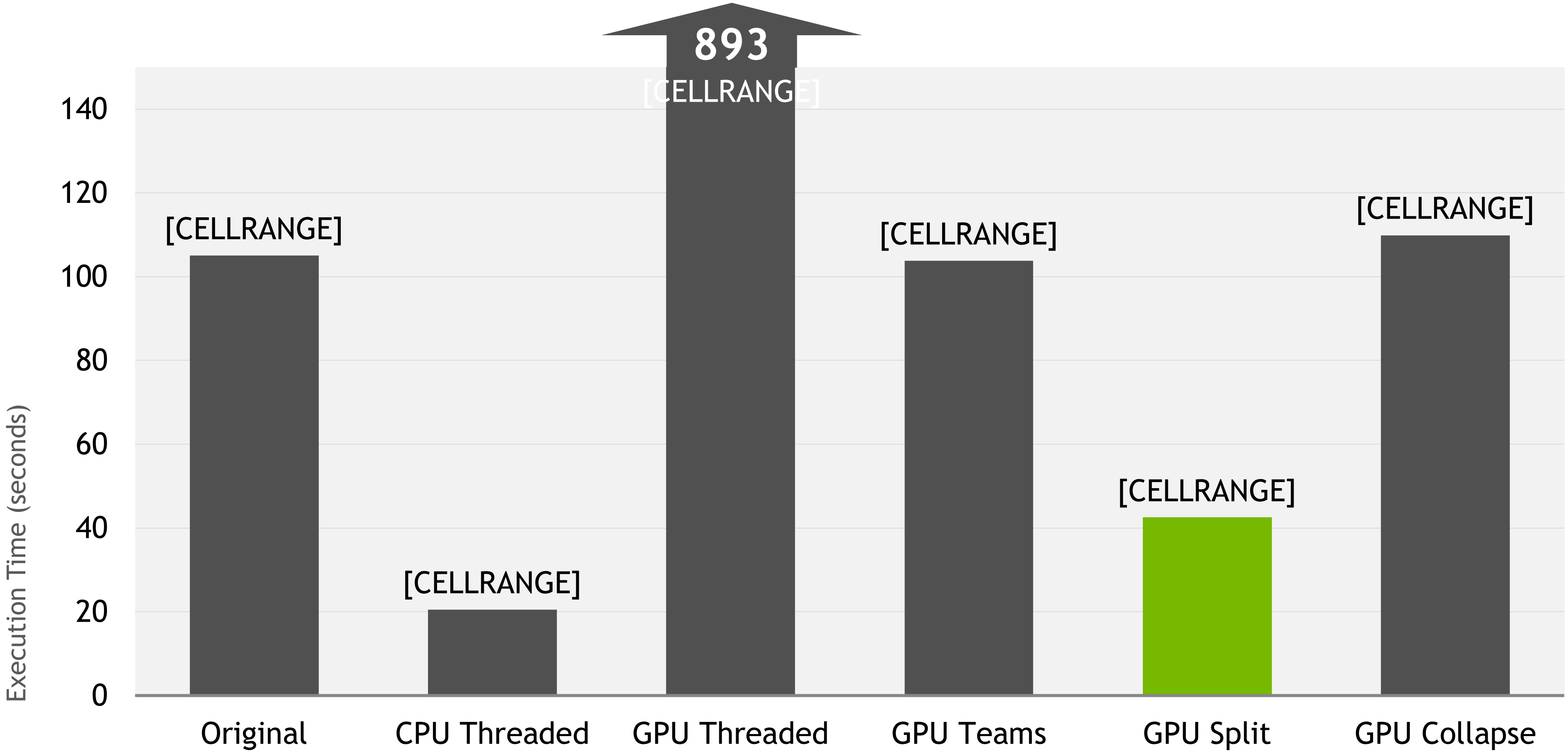
Workshare the “i” loop over threads.

Collapse

```
#pragma omp target teams distribute parallel for reduction(max:error) collapse(2)
        for( int j = 1; j < n-1; j++)
        {
            for( int i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }

#pragma omp target teams distribute parallel for collapse(2)
        for( int j = 1; j < n-1; j++)
        {
            for( int i = 1; i < m-1; i++ )
            {
                A[j][i] = Anew[j][i];
            }
        }
```

Collapse the two loops into one.

Execution Time (Smaller is Better)



NVIDIA Tesla K40, Intel Xeon E5-2690 v2 @ 3.00GHz

Improve Loop Scheduling

IMPROVE LOOP SCHEDULING

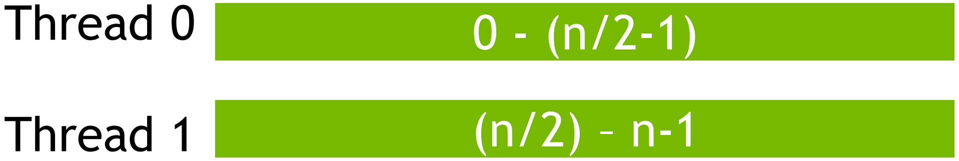
Most OpenMP compilers will apply a static schedule to workshared loops, assigning iterations in *N / num_threads* chunks.

- Each thread will execute contiguous loop iterations, which is very cache & SIMD friendly
- This is great on CPUs, but bad on GPUs

The SCHEDULE() clause can be used to adjust how loop iterations are scheduled.

EFFECTS OF SCHEDULING

!$OMP PARALLEL FOR SCHEDULE(STATIC)



Thread 0

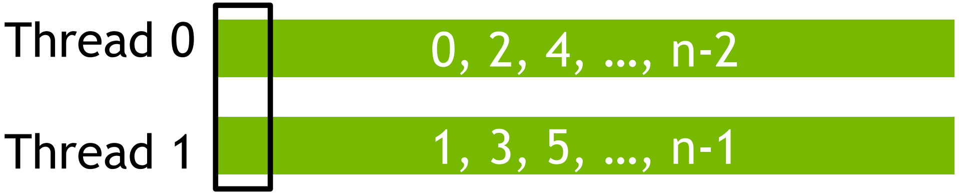
0 - (n/2-1)

Thread 1

(n/2) – n-1

Cache and vector friendly

!$OMP PARALLEL FOR SCHEDULE(STATIC,1)*



Thread 0

0, 2, 4, …, n-2

Thread 1

1, 3, 5, …, n-1

Memory coalescing friendly

*There’s no reason a compiler couldn’t do this for you.

Improved Schedule (Split)

```
#pragma omp target teams distribute
        for( int j = 1; j < n-1; j++)
        {
#pragma omp parallel for reduction(max:error) schedule(static,1)
            for( int i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }

#pragma omp target teams distribute
        for( int j = 1; j < n-1; j++)
        {
#pragma omp parallel for schedule(static,1)
            for( int i = 1; i < m-1; i++ )
            {
                A[j][i] = Anew[j][i];
            }
        }
```

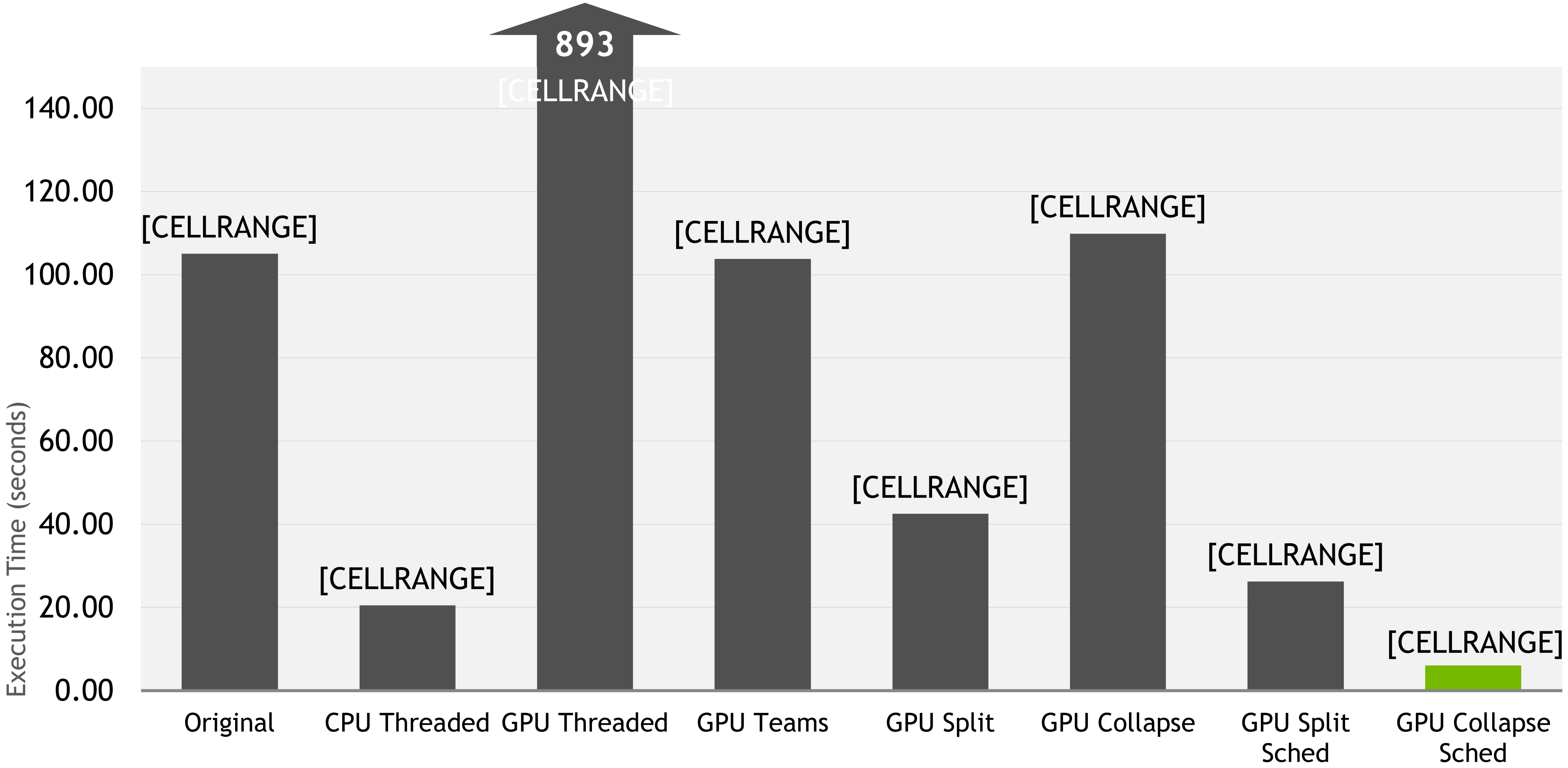
Assign adjacent threads adjacent loop iterations.

Improved Schedule (Collapse)

```
#pragma omp target teams distribute parallel for \
 reduction(max:error) collapse(2) schedule(static,1)
    for( int j = 1; j < n-1; j++)
    {
      for( int i = 1; i < m-1; i++ )
      {
        Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                  + A[j-1][i] + A[j+1][i]);
        error = fmax( error, fabs(Anew[j][i] - A[j][i]));
      }
    }

#pragma omp target teams distribute parallel for \
 collapse(2) schedule(static,1)
    for( int j = 1; j < n-1; j++)
    {
      for( int i = 1; i < m-1; i++ )
      {
        A[j][i] = Anew[j][i];
      }
    }
```

Assign adjacent threads adjacent loop iterations.

Execution Time (Smaller is Better)



NVIDIA Tesla K40, Intel Xeon E5-2690 v2 @ 3.00GHz

Additional Experiments

INCREASE THE NUMBER OF TEAMS

By default, CLANG will poll the number of SMs on your GPU and run that many teams of 1024 threads.

This is not always ideal, so we tried increasing the number of teams using the num_teams clause.

Test	SMs	2*SMs	4*SMs	8*SMs
A	1.00X	1.00X	1.00X	1.00X
B	1.00X	1.02X	1.16X	1.09X
C	1.00X	0.87X	0.94X	0.96X
D	1.00X	1.00X	1.00X	0.99X

DECREASED THREADS PER TEAM

CLANG always generate CUDA threadblocks of 1024 threads, even when the num_threads clause is used.

This number is frequently not ideal, but setting num_threads does not reduce the threadblock size.

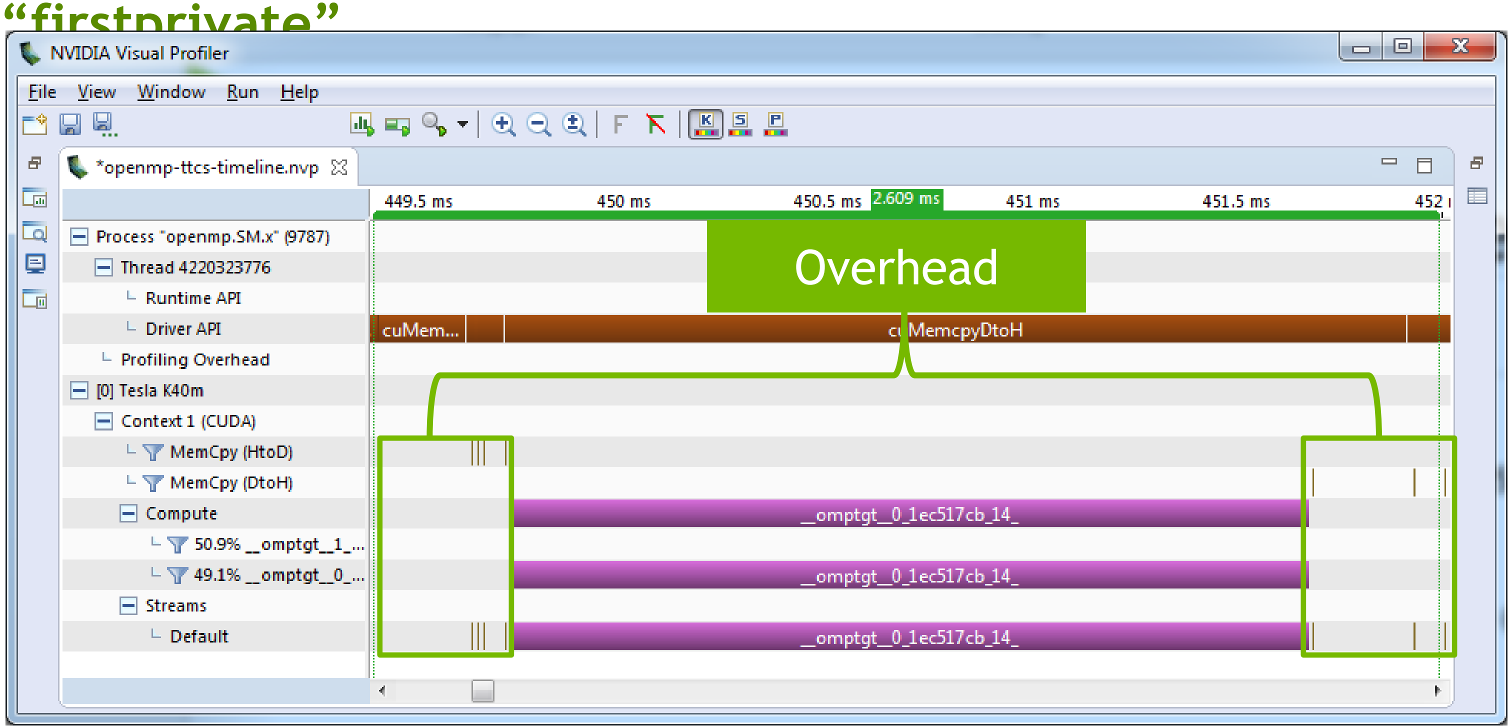
Ideally we'd like to use num_threads and num_teams to generate more, smaller threadblocks

We suspect the best performance would be collapsing, reducing the threads per team, and then using the remaining iterations to generate many teams, but are unable to do this experiment.

SCALAR COPY OVERHEAD

In OpenMP 4.0 scalars are implicitly mapped “tofrom”, resulting in very high overhead. Application impact: ~10%.

OpenMP4.5 remedied this by making the default behavior of scalars “firstprivate”



Note: In the meantime, some of this overhead can be mitigated by explicitly mapping your scalars “to”.

OPENACC & UNIFIED MEMORY

JACOBI ITERATION: OPENACC C CODE - CPU & GPU

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

#pragma acc parallel loop reduction(max:err)
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

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

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

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

  iter++;
}
```

Parallelize loop on accelerator

Parallelize loop on accelerator

Identify Parallelism

Express Parallelism

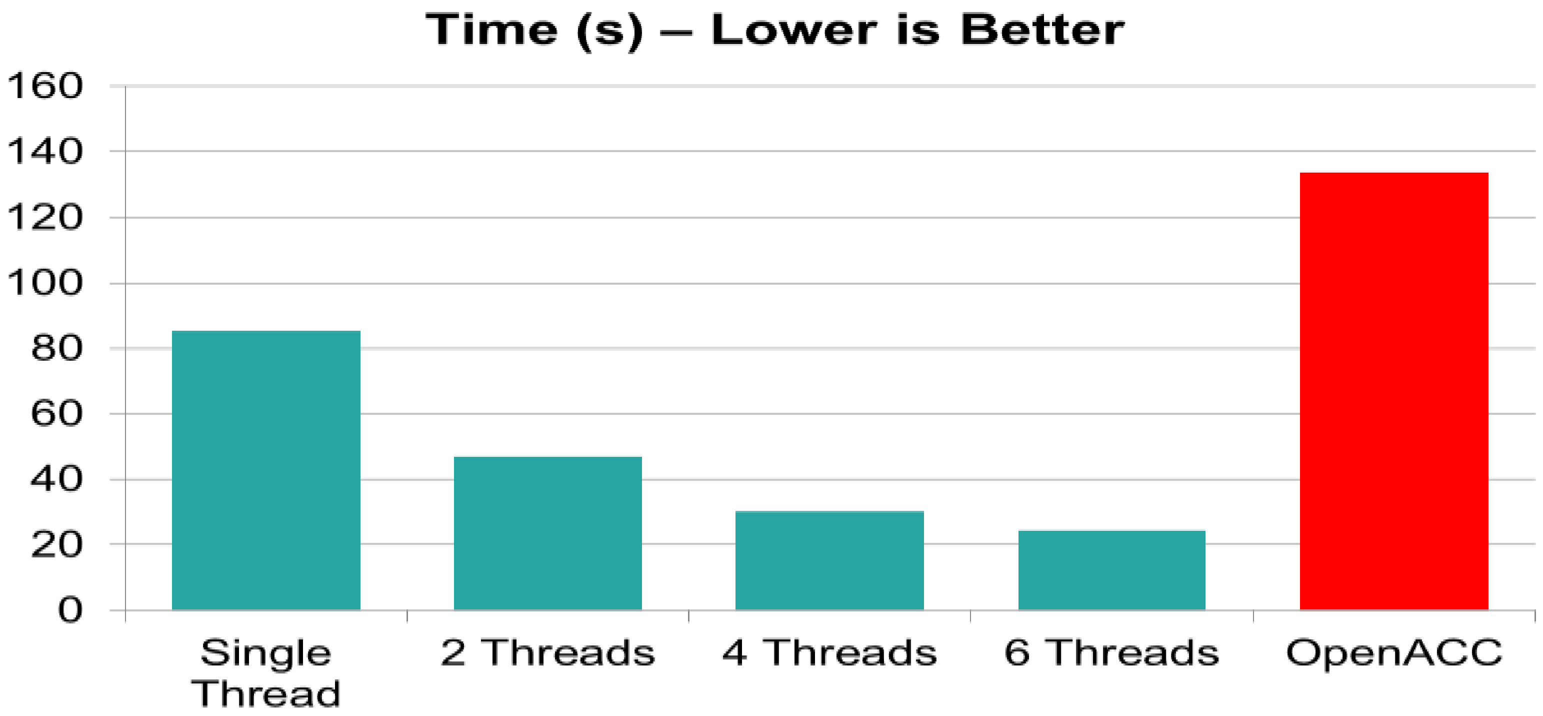
Express Data Locality

Optimize

BUILDING THE CODE

```
$ pgcc -acc -ta=nvidia:5.5,kepler -Minfo=accel -o laplace2d_acc laplace2d.c
main:
     56, Accelerator kernel generated
         57, #pragma acc loop gang /* blockIdx.x */
         59, #pragma acc loop vector(256) /* threadIdx.x */
     56, Generating present_or_copyout(Anew[1:4094][1:4094])
         Generating present_or_copyin(A[0:][0:])
         Generating NVIDIA code
         Generating compute capability 3.0 binary
     59, Loop is parallelizable
     63, Max reduction generated for error
     68, Accelerator kernel generated
         69, #pragma acc loop gang /* blockIdx.x */
         71, #pragma acc loop vector(256) /* threadIdx.x */
     68, Generating present_or_copyin(Anew[1:4094][1:4094])
         Generating present_or_copyout(A[1:4094][1:4094])
         Generating NVIDIA code
         Generating compute capability 3.0 binary
     71, Loop is parallelizable
```

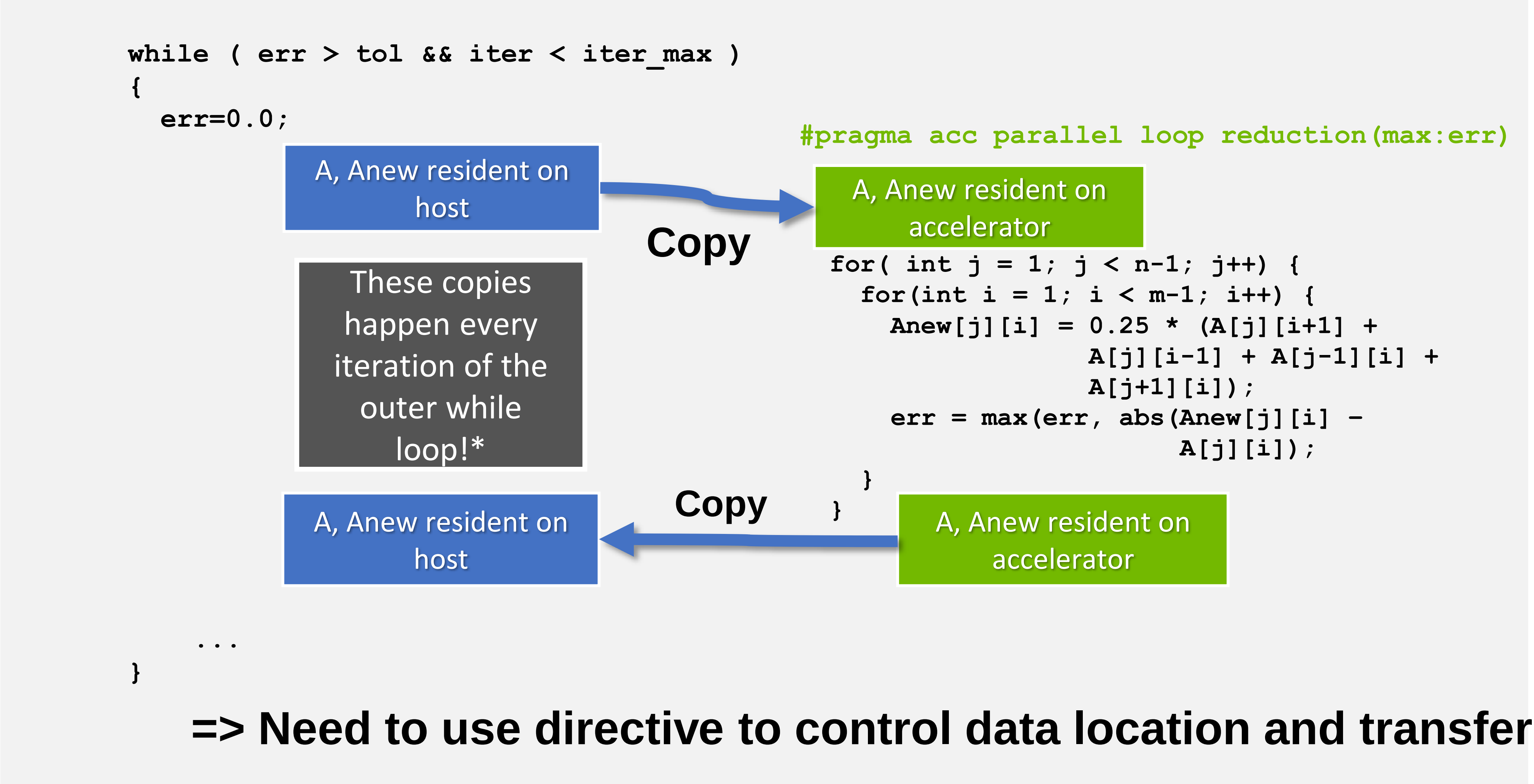
Why is OpenACC so much slower?



PROFILING AN OPENACC APPLICATION

```
$ nvprof ./laplace2d_acc
Jacobi relaxation Calculation: 4096 x 4096 mesh
==10619== NVPROF is profiling process 10619, command: ./laplace2d_acc
    0, 0.250000
  100, 0.002397
  200, 0.001204
  300, 0.000804
  400, 0.000603
  500, 0.000483
  600, 0.000403
  700, 0.000345
  800, 0.000302
  900, 0.000269
 total: 134.259326 s
==10619== Profiling application: ./laplace2d_acc
==10619== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 49.59%  44.0095s     17000  2.5888ms     864ns  2.9822ms  [CUDA memcpy HtoD]
 45.06%  39.9921s     17000  2.3525ms  2.4960us  2.7687ms  [CUDA memcpy DtoH]
  2.95%  2.61622s      1000  2.6162ms  2.6044ms  2.6319ms  main_56_gpu
  2.39%  2.11884s      1000  2.1188ms  2.1023ms  2.1374ms  main_68_gpu
  0.01%  12.431ms      1000  12.430us  12.192us  12.736us  main_63_gpu_red
```

Excessive Data Transfers



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

                                           #pragma acc parallel loop reduction(max:err)

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

    ...
}
```

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

These copies happen every iteration of the outer while loop!*

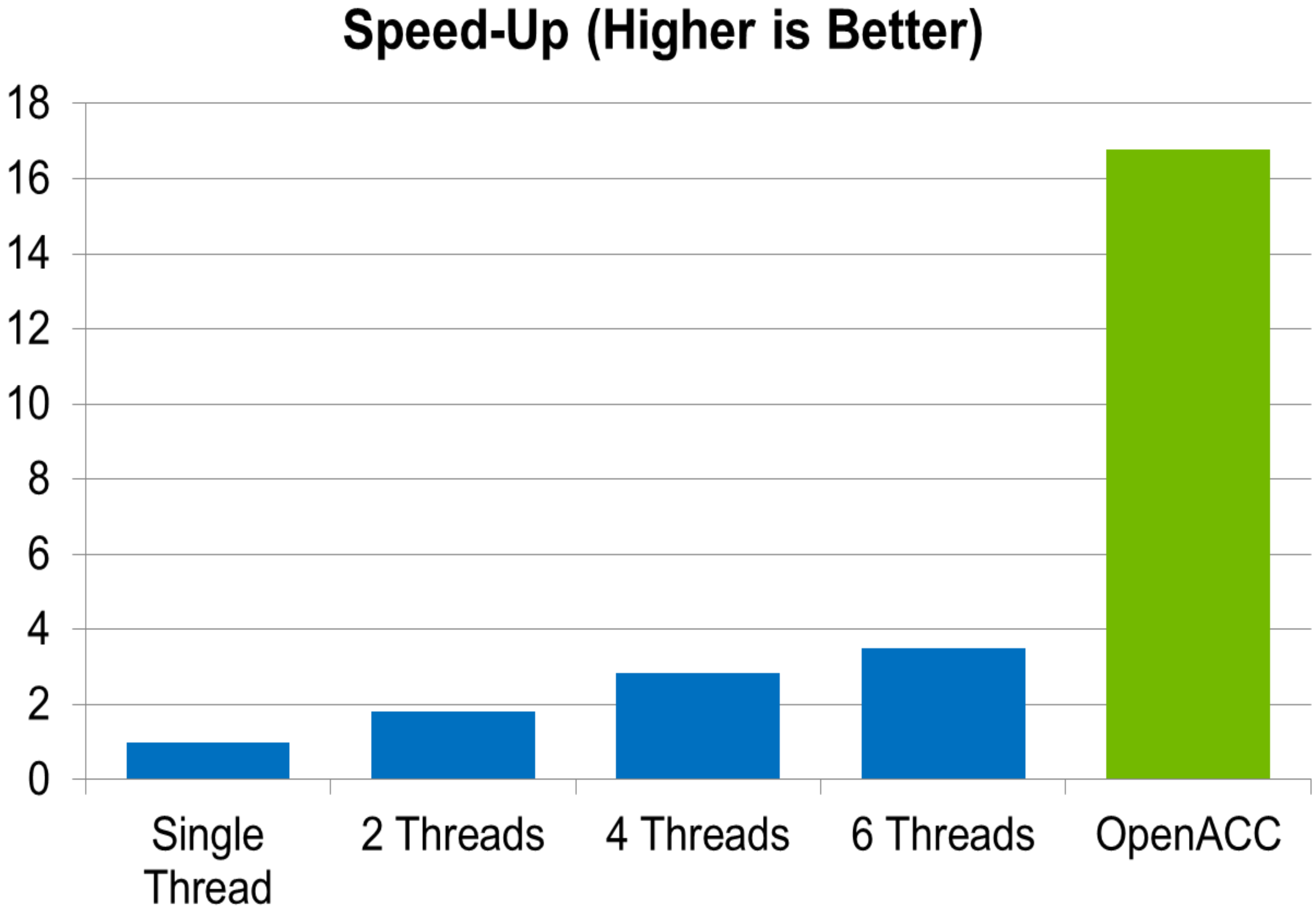
=> Need to use directive to control data location and transfers

Jacobi Iteration: OpenACC C Code

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

#pragma acc parallel loop reduction(max:err)
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

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

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

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

  iter++;
}
```

Copy A to/from the accelerator only when needed.
Create Anew as a device temporary.

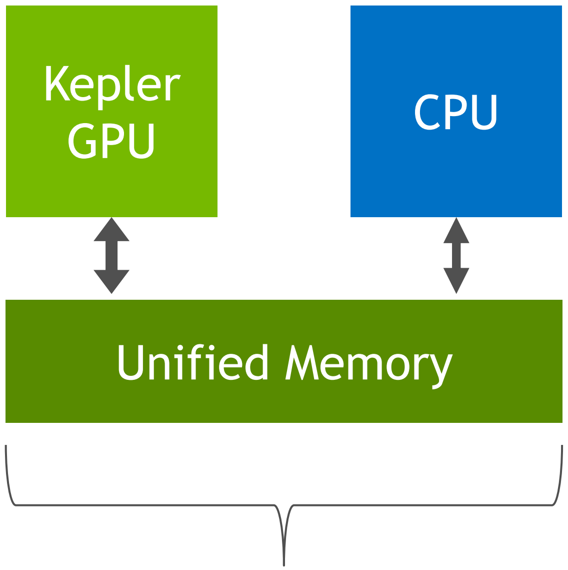
Identify Parallelism
Express Parallelism
Express Data Locality
Optimize



Speed-Up (Higher is Better)

KEPLER/MAXWELL UNIFIED MEMORY

CUDA 6+



Allocate Up To
GPU Memory Size

Simpler Programming & Memory Model

- Single allocation, single pointer, accessible anywhere
- Eliminate need for *explicit copy*
- Greatly simplifies code porting

Performance Through Data Locality

- Migrate data to accessing processor
- Guarantee global coherency
- Still allows explicit hand tuning

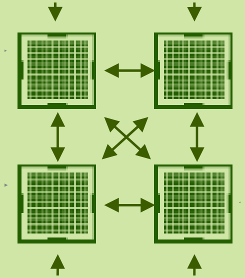
INTRODUCING TESLA P100

New GPU Architecture to Enable the World's Fastest Compute Node

Pascal Architecture

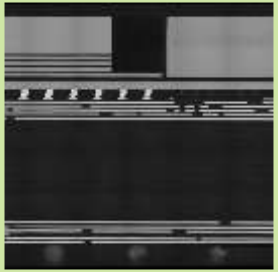
Highest Compute Performance

NVLink



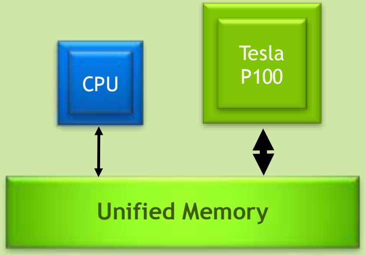
GPU Interconnect for Maximum Scalability

HBM2 Stacked Memory



Unifying Compute & Memory in Single Package

Page Migration Engine



Simple Parallel Programming with 512 TB of Virtual Memory

More P100 Features: compute preemption, new instructions, larger L2 cache, more…

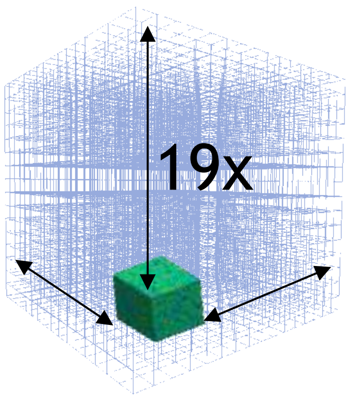
Find out more at http://devblogs.nvidia.com/parallelforall/inside-pascal
Pascal whitepaper at http://www.nvidia.com/object/pascal-architecture-whitepaper.html

PAGE MIGRATION ENGINE

Support Virtual Memory Demand Paging

- 49-bit Virtual Addresses
  - Sufficient to cover 48-bit CPU address + all GPU memory
- GPU page faulting capability
  - Can handle thousands of simultaneous page faults
- Up to 2 MB page size
  - Better TLB coverage of GPU memory

Unified Memory on Pascal enables simple programming with large datasets

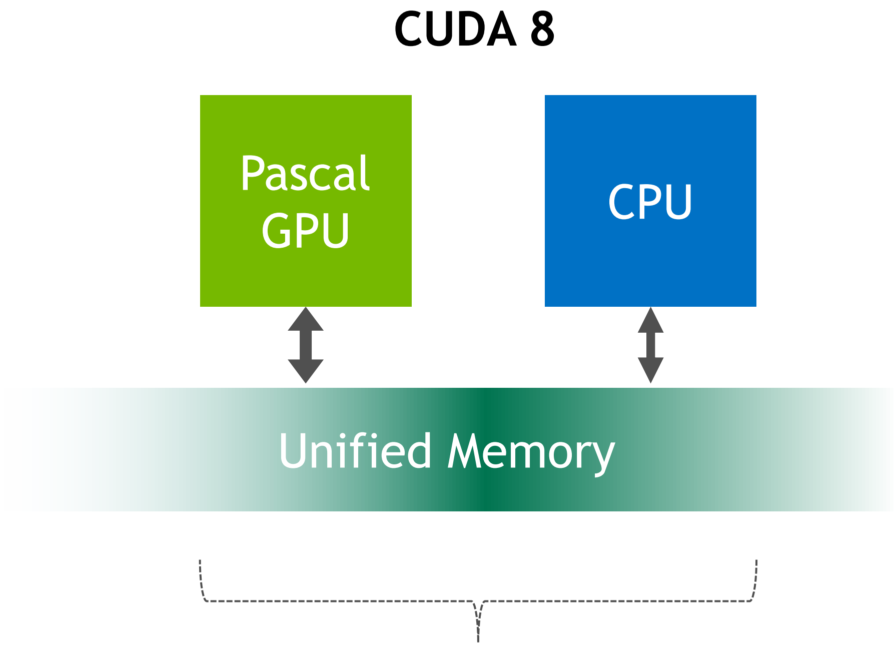


HPGMG with AMR

Larger Simulations &
More Accurate Results

PASCAL UNIFIED MEMORY

Large datasets, simple programming, High Performance



Allocate Beyond GPU Memory Size

Enable Large Data Models	Oversubscribe GPU memory Allocate up to system memory size
Tune Unified Memory Performance	Usage hints via cudaMemAdvise API Explicit prefetching API
Simpler Data Access	CPU/GPU Data coherence Unified memory atomic operations

UNIFIED MEMORY EXAMPLE

On-Demand Paging

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


void foo(int size) {
  char *data;
  cudaMallocManaged(&data, size);

  memset(data, 0, size);

  setValue<<<...>>>(data, size/2, 5);
  cudaDeviceSynchronize();

  useData(data);

  cudaFree(data);
}
```

Unified Memory allocation

Access all values on CPU

Access one value on GPU

HOW UNIFIED MEMORY WORKS IN CUDA 6

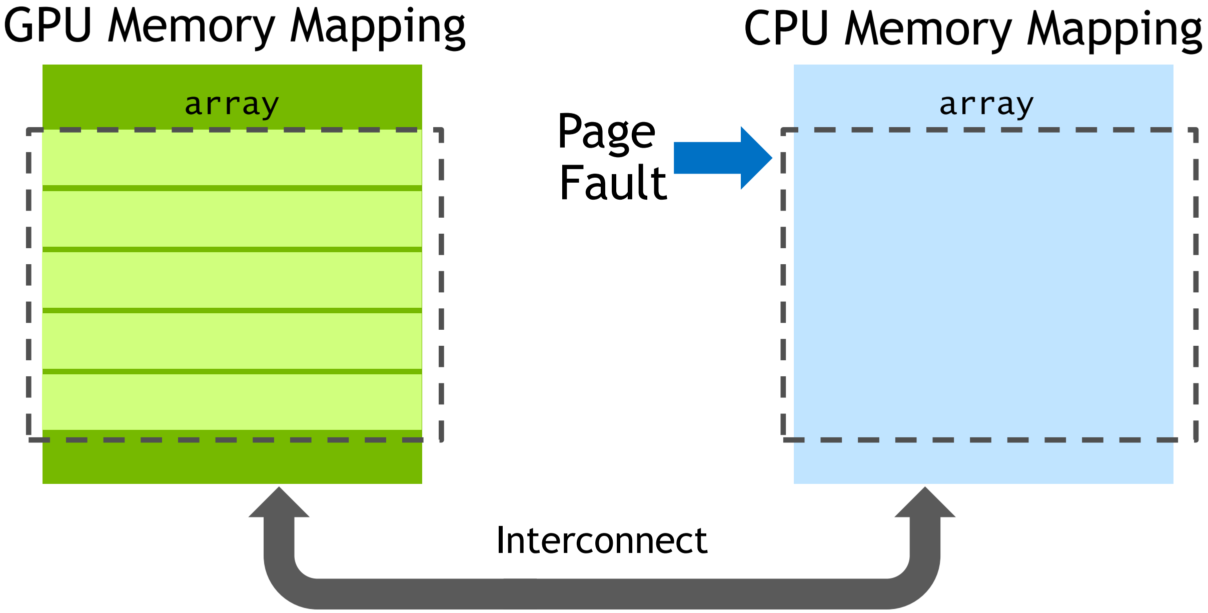
Servicing CPU page faults

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);
```



HOW UNIFIED MEMORY WORKS ON PASCAL

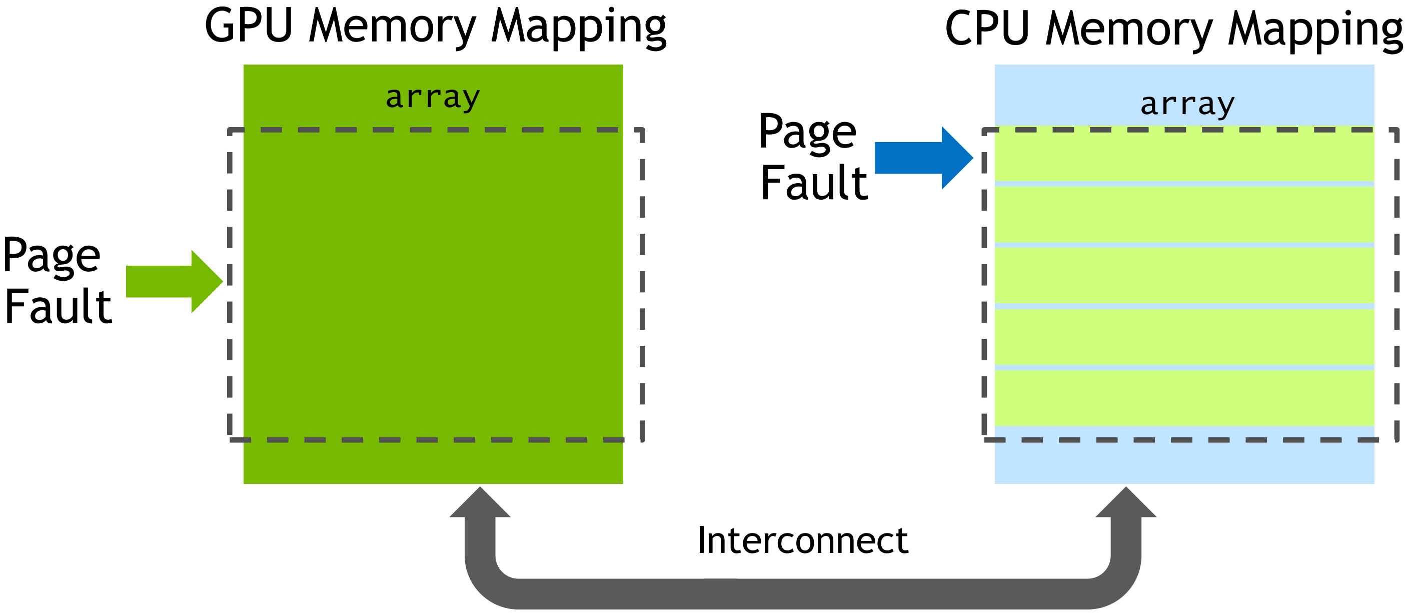
Servicing CPU *and* GPU Page Faults

GPU Code

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

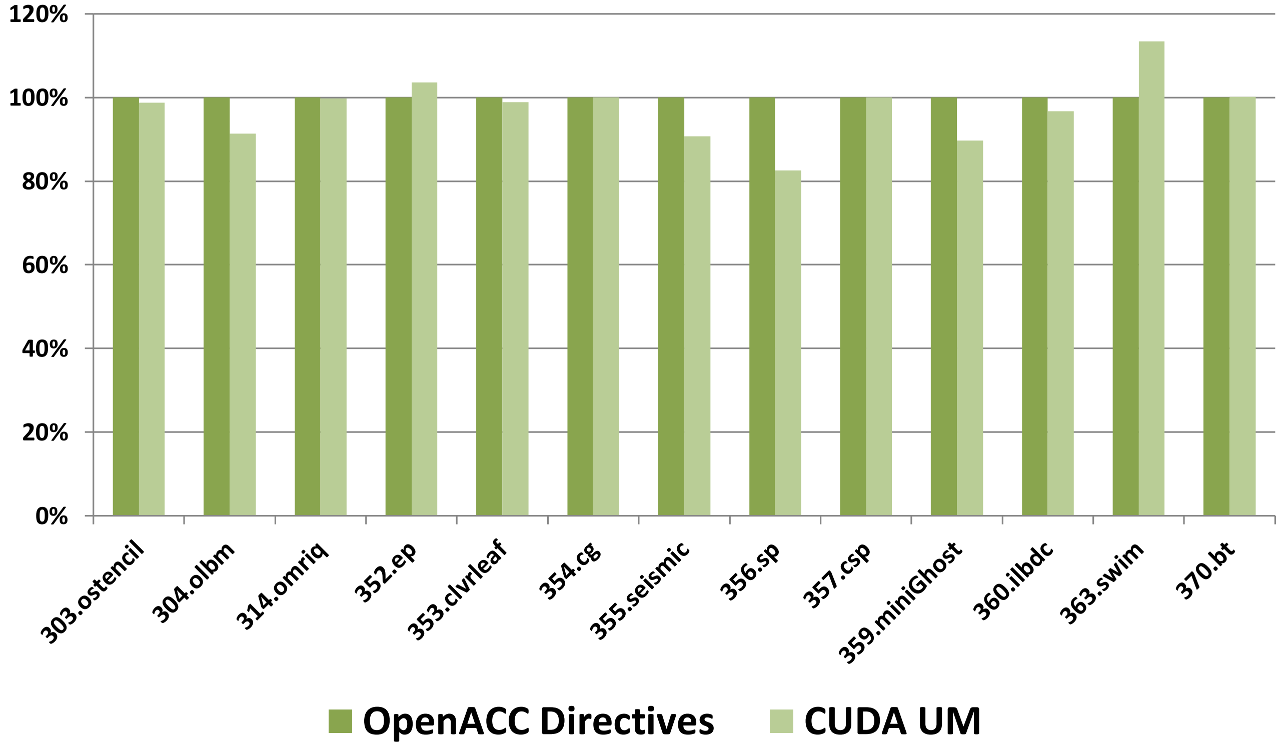
CPU Code

```
cudaMallocManaged(&array, size);

memset(array, size);

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



OPENACC AND CUDA UNIFIED MEMORY

PGI 15.1: OpenACC directive-based data movement vs OpenACC w/CUDA 6.5 Unified Memory on Kepler



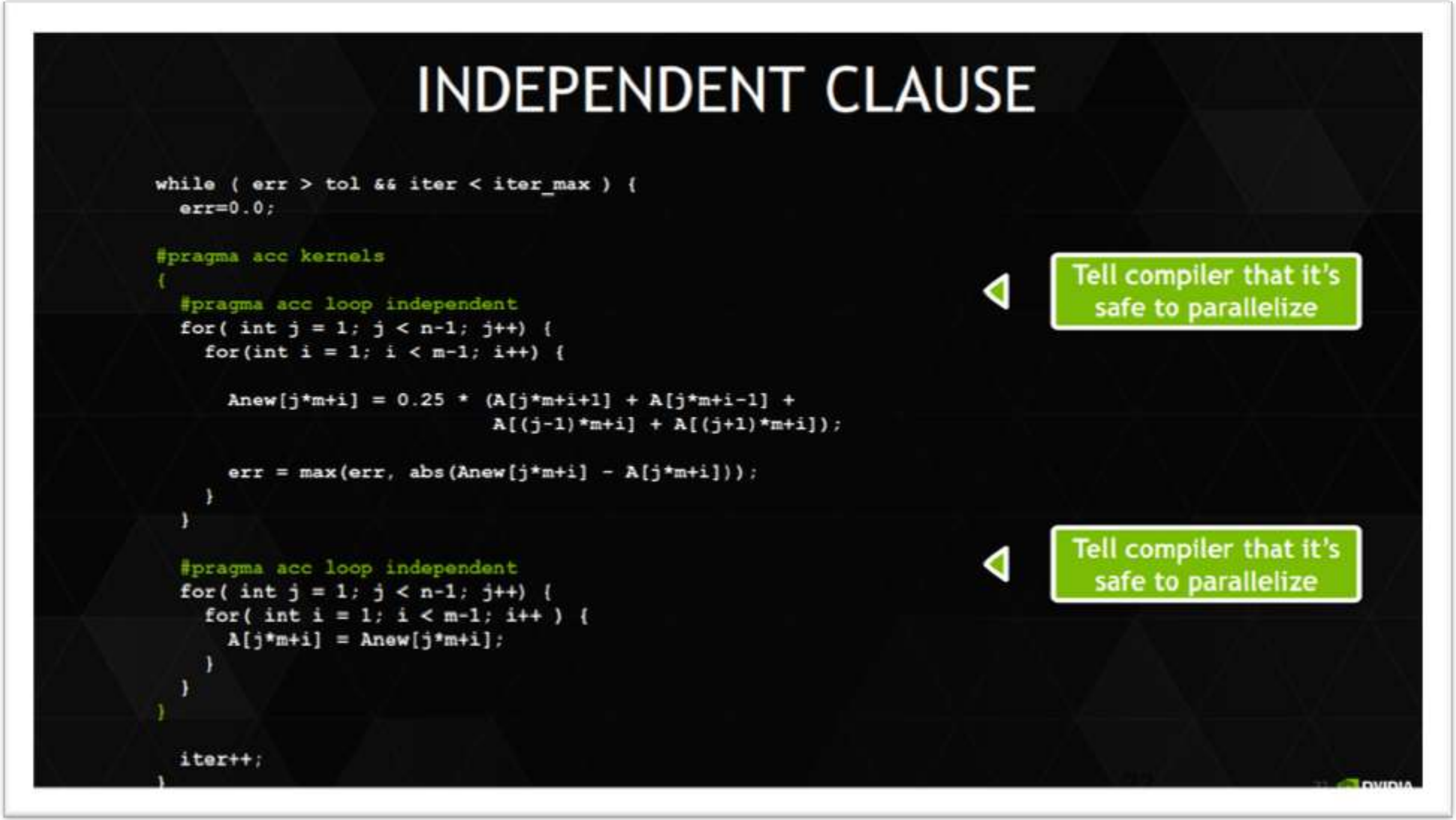
Features:

- Fortran ALLOCATE and C/C++ malloc/calloc/new can automatically use CUDA Unified Memory
- No explicit transfers needed for dynamic data (or allowed, for now)

Limitations:

- Supported only for dynamic data
- Program dynamic memory size is limited by UM data size
- UM data motion is synchronous
- Can be unsafe

OPENACC AND CUDA UNIFIED MEMORY



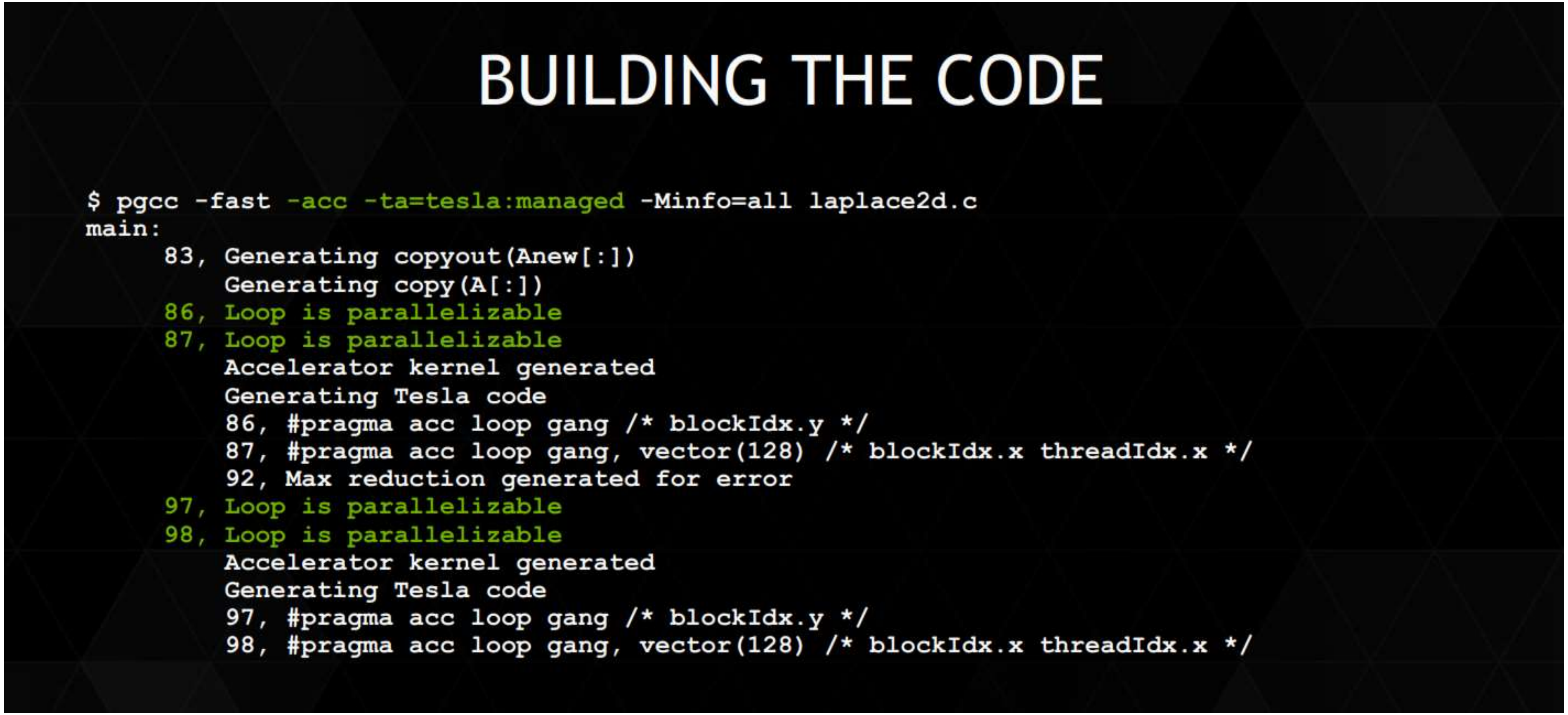
INDEPENDENT CLAUSE

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

#pragma acc kernels
{
  #pragma acc loop independent
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

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

      err = max(err, abs(Anew[j*m+i] - A[j*m+i]));
    }
  }

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

  iter++;
}
```

Tell compiler that it's safe to parallelize

Tell compiler that it's safe to parallelize

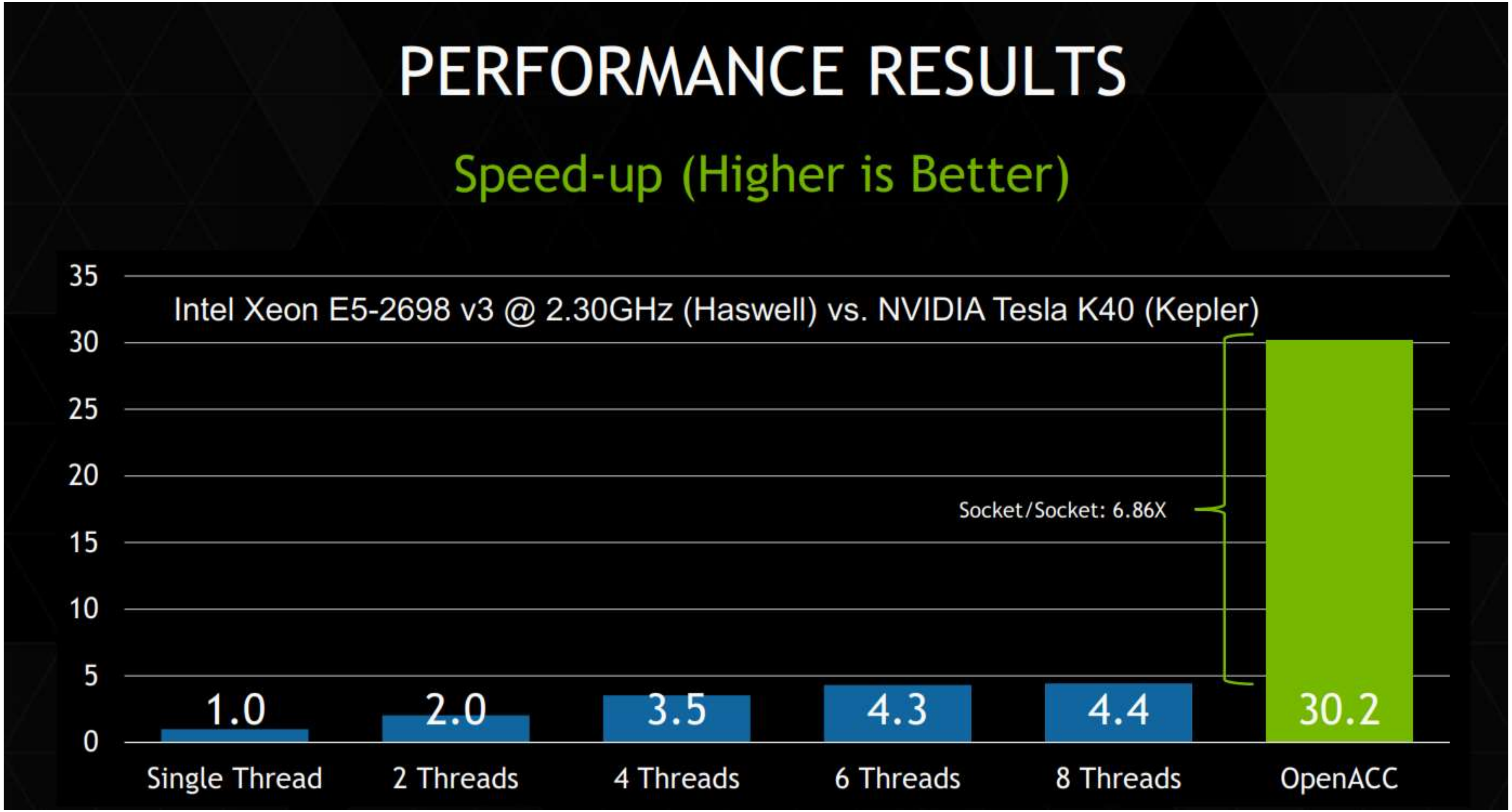
OPENACC AND CUDA UNIFIED MEMORY



BUILDING THE CODE

```
$ pgcc -fast -acc -ta=tesla:managed -Minfo=all laplace2d.c
main:
     83, Generating copyout(Anew[:])
         Generating copy(A[:])
     86, Loop is parallelizable
     87, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         86, #pragma acc loop gang /* blockIdx.y */
         87, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         92, Max reduction generated for error
     97, Loop is parallelizable
     98, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         97, #pragma acc loop gang /* blockIdx.y */
         98, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

OPENACC AND CUDA UNIFIED MEMORY



CONCLUSIONS

CONCLUSIONS

OpenMP & OpenACC, while similar, are still quite different in their approach

Each approach has clear tradeoffs with no clear *“winner”*

It should be possible to translate between the two, but the process may not be automatic

It is now possible to use OpenMP to program for GPUs, but the software is still very immature.

OpenMP for a GPU *will not* look like OpenMP for a CPU.

Performance will vary significantly depending on the exact directives you use. (149X in our example code)

ACKNOWLEDGEMENTS

James Beyer, NVIDIA

Jeff Larkin, NVIDIA

Nikolay Sakharnykh, NVIDIA

Questions: fcourteille@nvidia.com