Off-Loading using OpenACC

Advanced Computational Techniques
TCCM Erasmus Mundus course

Sept 26-30, 2022

University of Perugia

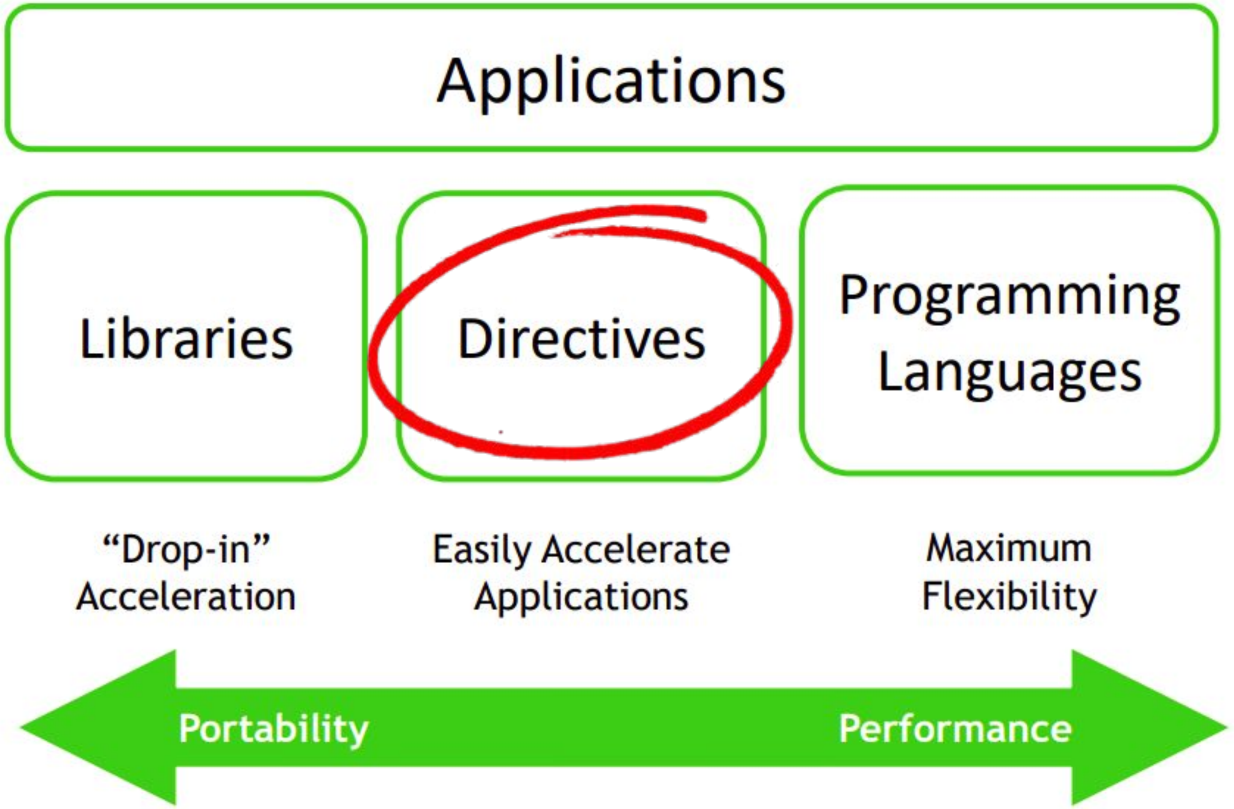
Sergio Orlandini
CINECA

s.orlandini@cineca.it

OpenACC Outline

- OpenACC Introduction
  - express parallelism
  - optimize data movement
  - practical examples
- Hands-On
  - Laplace 2D

3 Ways to Accelerate Applications



OpenACC Friendly Disclaimer

OpenACC does not make GPU programming easy.

(...)

GPU programming and parallel programming is not easy.

It cannot be made easy. However, GPU programming need not be difficult, and certainly can be made straightforward, once you know how to program and know enough about the GPU architecture to optimize your algorithms and data structures to make effective use of the GPU for computing.

OpenACC is designed to fill that role.

Michael Wolfe, The Portland Group

OpenACC

OpenACC is a specification of **compiler directives** and API routines for writing data parallel code in C, C++, or Fortran that can be compiled to parallel architectures, such as GPUs or multicore CPUs.

The goal of OpenACC is to provide a programming model that is **simple** to use and is performance **portable**, with which a **single source** code can be maintained between **different architectures**.
The OpenACC compiler generates the **kernels** that will be executed on different architectures.
High-level specification with compiler directives for expressing **parallelism for accelerators**, portable to a wide range of accelerators.
One specification for Multiple Vendors and Multiple Devices.

OpenACC History

- OpenACC specification was released in November 2011.
  - Original members: CAPS, Cray, Nvidia, Portland Group
- OpenACC 2.0 was released in June 2013
  - More functionality
  - Improve portability
- OpenACC 2.5 in November 2015
- OpenACC 2.7 in November 2018
- OpenACC 3.0 in November 2019
- OpenACC 3.1 in November 2020
- OpenACC 3.2 in November 2021

- OpenACC had more than 10+ member organizations

OpenACC Compilers

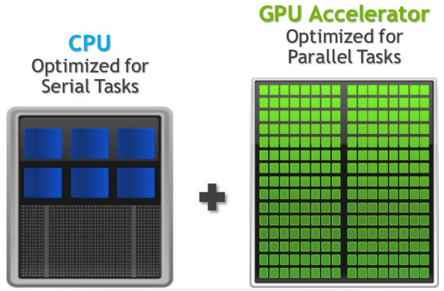
- **CRAY** provides full OpenACC 2.0 support from CCE 8.2
    - CCE only supports OpenACC for Fortran
    - C/C++ support was dropped in CCE 10.0
- **NVIDIA** HPC-SDK support to OpenACC 2.7 is almost complete
    - Support for OpenACC 2.0 starting from PGI 14.1
    - Support to OpenACC 2.5 is almost complete (starting from PGI 15.1)
    - hpc-sdk 22.3 supports almost all of the OpenACC 2.7 specification
- **GNU** supports OpenACC 2.6
    - A partial implementation in the 5.1 release
    - A dedicated branch for 7.1 realese for implementation of the OpenACC 2.0
    - Complete support of OpenACC 2.5 from 9.1 release
    - GCC 10 with support for OpenACC 2.6 on both NVIDIA and AMD platforms

OpenACC – Simple, Open, Portable, Powerful

```
main()
{

   <serial code>

   #pragma acc kernels
   //automatically runs on GPU
   {
      <parallel code>
   }
}
```



1. **Simple:**
   - Simple compiler directives
   - Directives are the easy path to accelerate compute intensive applications
   - Directives are added to serial source code
   - Mantaines portability of original code
   - Compiler parallelize code
2. **Open:**
   - Open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors
3. **Portable:**
   - Works on many-core GPUs and multi-core CPUs
4. **Powerful:**
   - GPU Directives allow complete access to the massive parallel power of a GPU

Directive Syntax

C/C++

```
#pragma acc directive [clause [,] clause] …]
```

Often followed by a structured code block

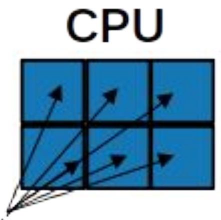
Fortran

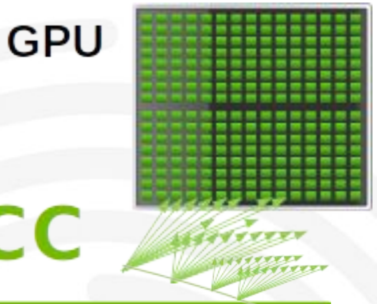
```
!$acc directive [clause [,] clause] …]
```

Often paired with a matching end directive surrounding a structured code block

```
!$acc end directive
```

OpenMP *vs* OpenACC





OpenMP

```
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

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

  #pragma acc parallel loop 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);
}
```

Porting to OpenACC

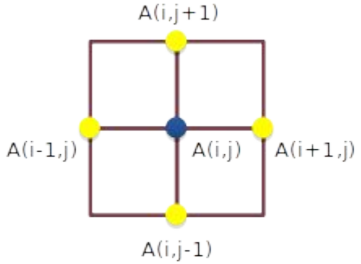
1. **Identify** available parallelism
2. Express **parallelism**
3. Express **data movement**
4. **Optimize** loop performance

Example: Jacobi Iteration

- Jacobi method is an iterative method to solve a system of linear equation
- It is a very common and useful algorithm
- Example: Jacobi method can be used to solve the Laplace differential equation in two variables (2D)
- The Laplace equation models the steady state of a function f defined in a physical 2D space where f is a physical quantity (e.g. Temperature)

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

- Iteratively converges to correct value by computing new values at each point from the average of neighboring points



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

Jacobi Iteration: C/C++ Code

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

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

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


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

    }
  }

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

  iter++;
}
```

1. Iterate until converged
2. Iterate across matrix elements
3. Calculate new value from neighbors
4. Compute max error for convergence
5. Swap input/output arrays

Jacobi Iteration: Fortran Code

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

  do j=1,m
    do i=1,n

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


       err = max(err, Anew(i,j) - A(i,j))

    end do
  end do

  do j=1,m-2
    do i=1,n-2
      A(i,j) = Anew(i,j)
    end do
  end do

  iter = iter +1
end do
```

1. Iterate until converged
2. Iterate across matrix elements
3. Calculate new value from neighbors
4. Compute max error for convergence
5. Swap input/output arrays

1. Identify Parallelism

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

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

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


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

    }
  }

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

  iter++;
}
```

Data dependency between iterations

Independent loop iterations

Independent loop iterations

Jacobi Iteration: OpenMP

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

#pragma omp parallel for collapse(2) 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 = max(error, abs(Anew[j][i] - A[j][i]);

    }
  }

#pragma omp 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];
    }
  }

  iter++;
}
```

`parallel` construct

- **Programmer identifies** a block of code suitable for parallelization
- and guarantees that **no dependency** occurs across iterations
- Compiler generates parallel instructions for that loop
  - e.g., a parallel CUDA kernel for a GPU

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

`kernels` construct

- The **`kernels`** construct expresses that a region **may** contain parallelism and the **compiler** determines what can be safely parallelized

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

kernel 1

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

kernel 2

```
!$acc end kernels
```

NB: The compiler identifies 2 parallel loops and 2 kernels are generated

`parallel` vs `kernels`

`parallel`

- Requires analysis by **programmer** to ensure safe parallelism
- Straightforward path from **OpenMP**
- Mandatory to fully control the different levels of parallelism
- Implicit barrier at the end of the parallel region

`kernels`

- **Compiler** performs parallel analysis and parallelizes what it believes safe
- Can cover **larger** area of code with a single directive
- Please, write clean codes and add directives to help the compiler
- Implicit barrier at the end and between each loop

`loop` construct

- Applies to a loop which must immediately follow this directive
- Describes:
  - type of parallelism
  - loop-private variables, arrays, and reduction operations
- We already encountered it combined with the parallel directive
  - combining kernels and loop is also possible but limits the capability of kernels construct (i.e. extending to wide regions of code)

C/C++

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

Fortran

```
!$acc loop [clause ...]
    structured block
!$acc end loop
```

Loop Reductions

- The `reduction` clause on a `loop` specifies a reduction operator on one or more scalar variables
    - For each variable, a private copy is created for each thread executing the associated loops
    - At the end of the loop, the values for each thread are combined using the reduction clause
- Reductions may be defined even at `parallel` level (advanced topic)
- Common operators are supported:

```
+   *   max   min   &&   ||  ....
```

2. Express Parallelism with OpenACC

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

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

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


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

    }
  }

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

  iter++;
}
```

2. Express Parallelism with OpenACC

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

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

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


      err = max(err, Anew(i,j) - A(i,j))

    end do
  end do

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

  iter = iter +1
end do
```

2. Express Parallelism with OpenACC

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

#pragma acc parallel loop 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 = max(error, 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++;
}
```

2. Express Parallelism with OpenACC

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

!$acc parallel loop reduction(max:error)
  do j=1,m
    do i=1,n

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


      err = max(err, Anew(i,j) - A(i,j))

    end do
  end do
!$acc end parallel loop

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

  iter = iter +1
end do
```

Jacobi Iteration: Compile

```
pgcc –acc -ta=tesla -Minfo=all –o laplace2d.x laplace2d.c
```

```
main:
     34, Loop not vectorized: may not be beneficial
         Unrolled inner loop 4 times
         Generated 3 prefetches in scalar loop
     44, Loop not vectorized/parallelized: potential early exits
     48, Generating copyout(Anew[1:4094][1:4094])
         Generating copyin(A[:][:])
         Generating copyout(A[1:4094][1:4094])
     50, Loop is parallelizable
     51, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         50, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         51, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
         54, Max reduction generated for error
     58, Loop is parallelizable
     59, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         58, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         59, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
```

Jacobi Iteration: Performance

Resources	Time / sec.
CPU 1 OpenMP thread	22.5
CPU 2 OpenMP threads	11.5
CPU 4 OpenMP threads	6.5
CPU 8 OpenMP threads	3.9
CPU 16 OpenMP threads	2.5

2 eight-core Intel(R) Xeon(R) CPU E5-2687W @ 3.10GHz, PCI-e 3.0

Jacobi Iteration: Performance

Resources	Time / sec.
CPU 1 OpenMP thread	22.5
CPU 2 OpenMP threads	11.5
CPU 4 OpenMP threads	6.5
CPU 8 OpenMP threads	3.9
CPU 16 OpenMP threads	2.5
OpenACC 1GPU	

2 eight-core Intel(R) Xeon(R) CPU E5-2687W @ 3.10GHz, PCI-e 3.0

GPU Nvidia Tesla K80

Jacobi Iteration: Performance

Resources	Time / sec.
CPU 1 OpenMP thread	22.5
CPU 2 OpenMP threads	11.5
CPU 4 OpenMP threads	6.5
CPU 8 OpenMP threads	3.9
CPU 16 OpenMP threads	2.5
OpenACC 1GPU	9.2

2 eight-core Intel(R) Xeon(R) CPU E5-2687W @ 3.10GHz, PCI-e 3.0

GPU Nvidia Tesla K80

Jacobi Iteration: What is going wrong?

```
Accelerator Kernel Timing data
laplace2d.c
  main
    69: region entered 1000 times
        time(us): total=77524918 init=240 region=77524678
                  kernels=4422961 data=66464916
        w/o init: total=77524678 max=83398 min=72025 avg=77524
        72: kernel launched 1000 times
            grid: [256x256]  block: [16x16]
            time(us): total=4422961 max=4543 min=4345 avg=4422
laplace2d.c
  main
    57: region entered 1000 times
        time(us): total=82135902 init=216 region=82135686
                  kernels=8346306 data=66775717
        w/o init: total=82135686 max=159083 min=76575 avg=82135
        60: kernel launched 1000 times
            grid: [256x256]  block: [16x16]
            time(us): total=8201000 max=8297 min=8187 avg=8201
        64: kernel launched 1000 times
            grid: [1]  block: [256]
            time(us): total=145306 max=242 min=143 avg=145
acc_init.c
  acc_init
    29: region entered 1 time
        time(us): init=158248
```

4.4 seconds → kernels=4422961

66.5 seconds → data=66464916

8.3 seconds → kernels=8346306

66.8 seconds → data=66775717

data construct

C/C++

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

Fortran

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

- Manages **explicitly** data movements between host and device
- Crucial to handle GPU data persistence
- Allows for decoupling the scope of GPU variables from that of the accelerated regions
- May be nested
- Data clauses define different possible behaviours

`data` **construct**

`copy ( list )`	**Allocates** memory on GPU and **copies** data from host to GPU when **entering** region and copies data to host when **exiting** region.
`copyin ( list )`	**Allocates** memory on GPU and **copies** data from host to GPU when **entering** region.
`copyout ( list )`	**Allocates** memory on GPU and **copies** data to the host when **exiting** region.
`create ( list )`	**Allocates** memory on GPU but does not copy.
`present ( list )`	Data is already **present** on GPU from another data region.

and `present_or_copy[in|out]`, `present_or_create`, `deviceptr`.

3. Express Data Movement

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

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

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


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

    }
  }

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

  iter++;
}
```

3. Express Data Movement

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

#pragma acc parallel loop 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 = max(error, 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++;
}
```

Jacobi Iteration: Performance

Resources	Time / sec.
CPU 1 OpenMP thread	22.5
CPU 2 OpenMP threads	11.5
CPU 4 OpenMP threads	6.5
CPU 8 OpenMP threads	3.9
CPU 16 OpenMP threads	2.5
OpenACC 1GPU	

2 eight-core Intel(R) Xeon(R) CPU E5-2687W @ 3.10GHz, PCI-e 3.0

GPU Nvidia Tesla K80

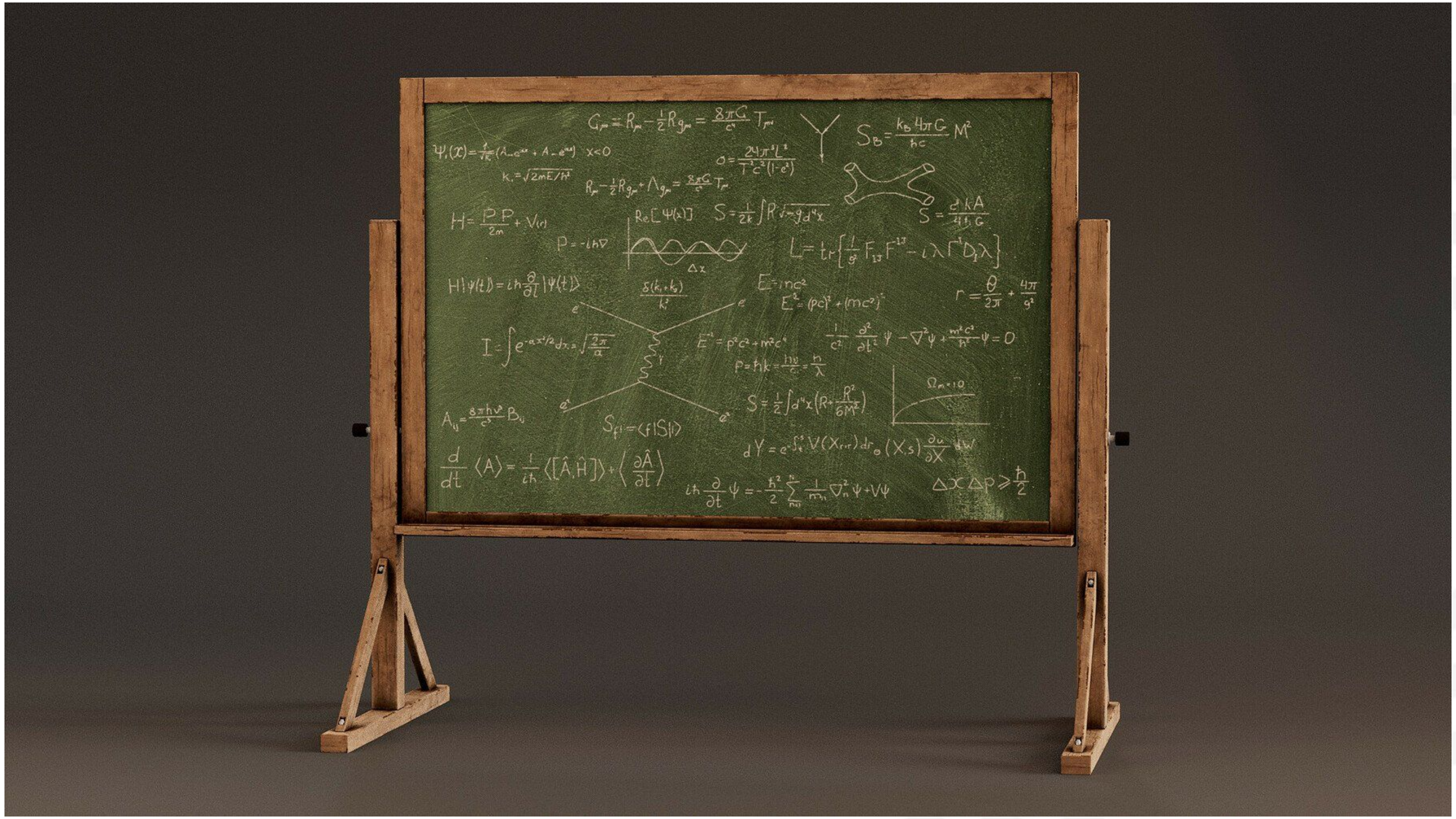
Jacobi Iteration: Performance

Resources	Time / sec.
CPU 1 OpenMP thread	22.5
CPU 2 OpenMP threads	11.5
CPU 4 OpenMP threads	6.5
CPU 8 OpenMP threads	3.9
CPU 16 OpenMP threads	2.5
OpenACC 1GPU	**0.6**

2 eight-core Intel(R) Xeon(R) CPU E5-2687W @ 3.10GHz, PCI-e 3.0

GPU Nvidia Tesla K80

Hands-on



Hands-on

Laplace 2D

- Compile serial code for reference
- Accelerate serial code with **OpenACC**
- Use `kernels` construct
- Use `parallel` construct
- Optimize data movements with `data` construct
- Performance