Introduction to OpenACC
16 May 2013
GPUs Reaching Broader Set of Developers

- Universities
- Supercomputing Centers
- Oil & Gas

- CAE
- CFD
- Finance
- Rendering
- Data Analytics
- Life Sciences
- Defense
- Weather
- Climate
- Plasma Physics

Early Adopters

- Research

Time

1,000,000's

100,000's

2004

Present
3 Ways to Accelerate Applications

- Libraries
  - “Drop-in” Acceleration

- OpenACC Directives
  - Easily Accelerate Applications

- Programming Languages
  - Maximum Flexibility
Introducing OpenACC
The Standard for GPU Directives

- OpenACC allows programmers to provide simple hints, known as “directives” to the compiler.

- These directives identify which areas of code to accelerate, without requiring programmers to modify or adapt the underlying code itself.

- By exposing parallelism to the compiler, directives allow the compiler to do the detailed work of mapping the computation onto the accelerator.
Advantages of OpenACC

**Easy:** OpenACC directives supply an easy path to accelerate compute intensive applications.
  • Modelled on the familiar OpenMP directives format.

**Open:** OpenACC is an open GPU directives standard, making GPU programming straightforward and portable.
  • Currently Nvidia GPUs and multicore CPUs. More to come…

**Powerful:** GPU Directives allow complete access to the massively parallel power of a GPU.
High-Level Language...

- Compiler directives to specify parallel regions in C & Fortran.
  - Offload parallel regions.
  - Portable across OSes, host CPUs, accelerators, and compilers.

- Create high-level heterogeneous programs.
  - Without explicit accelerator initialization.
  - Without explicit data or program transfers between host and accelerator.
...Low-Level Access

- Programming model allows programmers to start simple.
  - Compiler gives additional guidance.

- Loop mappings, data location, and other performance details.

- Compatible with other GPU languages and libraries.
  - Interoperate between CUDA C/Fortran and GPU libraries.
  - e.g. CUFFT, CUBLAS,CUSPARSE, etc.
OpenACC Directives Overview

Program myscience
... serial code ...
!$acc kernels
do k = 1,n1
do i = 1,n2
... parallel code ...
enddo
enddo
!$acc end kernels
...
End Program myscience

Simple Compiler hints
Compiler Parallelises code
Works on many-core GPUs & multicore CPUs

Your original Fortran or C code
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);
}

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);
}
Benefits of CUDA vs Time to Implement

- CUDA C/C++ or Fortran are powerful techniques, offering dramatic performance increases.

- While it has become increasingly user friendly, there are many programmers who can't afford the time to learn and apply a parallel programming language.

- Scientists and engineers also work with huge existing code bases and can only make minor changes to their code that are portable across hardware and operating systems.
Directives: Easy & Powerful

Real-Time Object Detection
Global Manufacturer of Navigation Systems

Valuation of Stock Portfolios using Monte Carlo
Global Technology Consulting Company

Interaction of Solvents and Biomolecules
University of Texas at San Antonio

5x in 40 Hours

2x in 4 Hours

5x in 8 Hours
Local Success with OpenACC

The University of Melbourne
Department of Zoology
Professor Kerry Black

65x in 48 hours

Better understand complex reasons by lifecycles of snapper fish in Port Phillip Bay

* Achieved using the PGI Accelerator Compiler
OpenACC Specification and Website

- Full OpenACC 1.0 Specification available online.
- Public Comment Draft of 2.0 Specification now available online.
- Quick reference card also available.
- Beta implementations available now from PGI, Cray, and CAPS.

www.openacc.org
How does it work?

- Programmers provide simple hints to the compiler.
- In C/C++ these hints are implemented using the `#pragma acc` directive (`!$acc` in Fortran).

The ‘acc’ identifier directives tell the compiler to enable accelerator functions during the compilation of your program code.

A fully defined accelerator directive should look as follows:

```
#pragma acc directive-name [clause [,,, clause]...]
```
OpenACC Directives

There directive-name specifies the type of accelerator action you want to perform.

For example, #pragma acc kernels is an OpenACC directive that compiles a region of your program into a sequence of kernels to execute on your GPU device.

The clause provides specific instructions for the chosen directive.

For example, #pragma acc kernels copyin(A[0:n]) declares that array A has values that need to be copied to the device memory.
OpenACC Directives

- There **directive-name** specifies the type of accelerator action you want to perform.

- For example, `#pragma acc kernels` is an OpenACC directive that compiles a region of your program into a sequence of kernels to execute on your GPU device.

- The **clause** provides specific instructions for the chosen directive.

  For example, `#pragma acc kernels copyin(A[0:n])` declares that array A has values that need to be copied to the device memory.

**NB:** This is NOT array ‘slice’ notation (Python, MATLAB, Fortran users beware!!) This means start at idx 0 and copy the following n elements
Example: Vector Addition Serial Code

```c
int main ()
{
    ...serial code...

    for( i=0; i < n; i++) {
        c[i] = a[i] + b[i]
    }

    ...serial code...
}
```
Vector Addition Device Code (CUDA)

Launch a CUDA kernel for the Vector addition

```c
__global__ void vecaddgpu(float *r, float *a, float *b, int n)
{
    // Get global thread ID
    int id = blockIdx.x*blockDim.x+threadIdx.x;

    // Make sure we do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}
```
Vector Addition Device Code (OpenACC)

Accelerator Kernel launched with #pragma acc

```c
void vecaddgpu( float *restrict r, float *a, float *b, int n )
{
    //Launch GPU Accelerator Kernel
    #pragma acc kernels copyin(a[0:n],b[0:n]) copyout(c[0:n])
    for( int i = 0; i < n; ++i ) {
        c[i] = a[i] + b[i];
    }
}
```
Device Code: What’s Similar?

- Both techniques require an accelerator (i.e. GPU) kernel to be launched on the device.

- The host variables (e.g. arrays) need to be copied from the host memory to the device memory.

- Values in the device memory need to be copied back to the host memory at the end of the accelerator region.
Device Code: What’s Changed?

- `#pragma acc` directive replaces `__global__` as the GPU kernel generator.

- Variables are copied from host to device (and vice versa) using `#pragma acc (copyin() & copyout())` directive clauses instead of using `cudaMemcpy()` in the host code.

- Thread id allocation (`blockIdx.x*blockDim.x+threadIdx.x`) is handled ‘behind the scenes’ by OpenACC.

- A `restrict` keyword is placed on array ‘r’ (explained later).
OpenACC parallel vs. kernels

**PARALLEL**
- Requires analysis by programmer to ensure safe parallelism
- Straightforward path from OpenMP

**KERNELS**
- Compiler performs parallel analysis and parallelizes what it believes safe
- Can cover larger area of code with single directive

Both approaches are equally valid and can perform equally well.
Kernels Construct

Each loop is executed as a separate kernel on the GPU.

```
#pragma acc kernels
    for (i=0; i<n; i++){
        a(i) = 0.0
        b(i) = 1.0
        c(i) = 2.0
    }

    for (i=0; i<n; i++){
        a(i) = b(i) + c(i)
    }
```

The two kernels will execute synchronously by default.
Kernels Clauses

- **if( condition )**
  Generates two copies of the construct (host and device) and executes one of the copies when an IF condition is reached.

- **async [( scalar-integer-expression )]**
  The kernels region will be executed by the accelerator device asynchronously while the host process continues with the code following the region.

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

- **copyin( list )**
  Allocates GPU memory and copies data from the host when entering the region.
**Kernels Clauses**

- `copyout( list )`
  Allocates host memory copies data to the host when exiting the region.

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

Other clauses include:
- `present( list )`
- `present_or_copy( list )`
- `present_or_copyin( list )`
- `present_or_copyout( list )`
- `present_or_create( list )`
- `deviceptr( list )`

For more information on clauses:
[http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf](http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf)
The restrict keyword

- Applied to a pointer. For example:
  ```c
  float* restrict r
  ```

- Without the `restrict` keyword, pointer aliasing may occur, whereby the same memory location can be accessed using different names.

- OpenACC compilers often require the `restrict` keyword to determine independence of memory locations.
  - Otherwise the compiler can’t parallelize loops that access `r`.

http://en.wikipedia.org/wiki/Restrict
Vector Addition Host Code (CUDA)

```c
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

// CUDA kernel. Each thread takes care of one element of c
__global__ void vecaddgpu(float *d_a, float *d_b, float *d_c, int n)
{
    // Device Code
}

int main(int argc, char* argv[]) {
    // Size of vectors
    int n = 100000;
    float *h_a; 
    float *h_b; 
    float *h_c; 
    float *d_a; 
    float *d_b; 
    float *d_c;
    // Allocate memory for each vector on host
    size_t bytes = n*sizeof(float);
    h_a = (float*)malloc(bytes);
    h_b = (float*)malloc(bytes);
    h_c = (float*)malloc(bytes);
    // Allocate memory for each vector on GPU
    cudaMemcpy(d_a, h_a, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, bytes, cudaMemcpyHostToDevice);
    // Initialise vectors on host
    int i;
    for(i = 0; i < n; i++) {
        h_a[i] = sinf(i)*sinf(i);
        h_b[i] = cosf(i)*cosf(i);
    }
    // Copy host vectors to device
    cudaMemcpy(d_a, h_a, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, bytes, cudaMemcpyHostToDevice);
    // Execute the kernel
    int blockSize, gridSize;
    blockSize = 1024;
    gridSize = (int)ceil((float)n/blockSize);
    vecaddgpu<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);
    // Copy result vector from device to host
    cudaMemcpy(h_c, d_c, bytes, cudaMemcpyDeviceToHost);
    // Sum up vector c and print result divided by n, this should equal 1 within error
    float sum = 0;
    for(i = 0; i < n; i++)
        sum += h_c[i];
    printf("final result: %f\n", sum/n);
    // Release device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    // Release host memory
    free(h_a);
    free(h_b);
    free(h_c);
    return 0;
}
```

- **Device kernel**
  ```c
  __global__ void vecAddGpu(...)
  ```

- Explicitly define separate host `h_x` and device `d_x` vectors

- Allocate host vector memory using `malloc()`

- Allocate device vector memory using `cudaMalloc()`

- Initialise vectors on host (**for loop**)

- Copy initialised values to device using `cudaMemcpy()`

- Launch kernel to do work `vecaddgpu<<< ... >>>`

- Copy result vector from device to host using `cudaMemcpy()`

- Free device memory using `cudaFree()`

- Free host memory using `free()`
What is one of the most significant differences between host-only and combined host+accelerator based programs?

**MEMORY**

In accelerator programming languages such as CUDA, data movement between the memories can dominate the user’s host code.

- In the OpenACC model, data movement between the memories is implicit and managed by the compiler.
- Controlled by the directives from the programmer (e.g. `copyin(list)`)

**Host Code: OpenACC**
Host Code: OpenACC

While less code-intensive, programmers must be aware of the potentially separate memories for reasons including (but not limited to):

- **Memory bandwidth** between host and device, which determines the compute intensity required to accelerate a region of code.

- **Device memory size**, which can prohibit offloading regions of code that operate on very large amounts of data.
Vector Addition Host Code (OpenACC)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// CUDA kernel. Each thread takes care of one element of c
void vecaddgpu(float *d_a, float *d_b, float *restrict d_c, int n)
{
#pragma acc kernels copyin(d_a[0:n],d_b[0:n]) copyout(d_c[0:n])
for (int i = 0; i < n; ++i)
    d_c[i] = d_a[i] + d_b[i];
}

int main(int argc, char* argv[]) {
    // Size of vectors
    int n = 100000;
    float *h_a;
    float *h_b;
    float *h_c;
    float *d_a;
    float *d_b;
    float *d_c;
    // Allocate memory for each vector on host
    size_t bytes = n*sizeof(float);
    h_a = (float*)malloc(bytes);
    h_b = (float*)malloc(bytes);
    h_c = (float*)malloc(bytes);
    // Allocate memory for each vector on GPU
    cudaMalloc(&d_a, bytes);
    cudaMalloc(&d_b, bytes);
    cudaMalloc(&d_c, bytes);
    // Initialize vectors on host
    int i;
    for (i = 0; i < n; i++) {
        h_a[i] = sinf(i)*sinf(i);
        h_b[i] = cosf(i)*cosf(i);
    }
    // Copy host vectors to device
cudaMemcpy(d_a, h_a, bytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, bytes, cudaMemcpyHostToDevice);
    // Execute the kernel
    int blockSize, gridSize;
    blockSize = 1024;
    gridSize = (int)(ceil((float)n/blockSize));
    vecaddgpu<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);
    // Copy result from device to host using cudaMemcpy()
    cudaMemcpy(h_c, d_c, bytes, cudaMemcpyDeviceToHost);
    // Free device memory using cudaFree()
    free(h_a);
    free(h_b);
    free(h_c);
    return 0;
}
```

Remove `__global__` and replace device code with

```
#pragma acc kernels copyin(...) copyout(...)
```

Define host `h_x` and device `d_x` vectors

Allocate host vector memory using `malloc()`

Allocate device vector memory using `cudaMalloc()`

Initialise vectors on host (`for` loop)

Copy initialised values to device using `cudaMemcpyncp()`

Launch kernel to do work `vecaddgpu<<< ... >>>`

Copy result vector from device to host using `cudaMemcpyncp()`

Free device memory using `cudaFree()`

Free host memory using `free()`
Vector Addition Host Code (OpenACC)

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// CUDA kernel. Each thread takes care of one element of c
void vecaddgpu(float *a, float *b, float *restrict c, int n) {
    #pragma acc kernels copyin(a[0:n],b[0:n]) copyout(c[0:n])
    for (int i = 0; i < n; ++i)
        c[i] = a[i] + b[i];
}

int main(int argc, char* argv[]) {
    // Size of vectors
    int n = 100000;
    float *a;
    float *b;
    float *c;

    // Allocate memory for each vector on host
    size_t bytes = n*sizeof(float);
    a = (float*)malloc(bytes);
    b = (float*)malloc(bytes);
    c = (float*)malloc(bytes);

    // Initialize vectors on host
    int i;
    for (i = 0; i < n; i++) {
        a[i] = sinf(i)*sinf(i);
        b[i] = cosf(i)*cosf(i);
    }

    // Execute the kernel
    vecaddgpu(a, b, c, n);

    // Sum up vector c and print result divided by n, this should equal 1 within error
    float sum = 0;
    for (i = 0; i < n; ++i)
        sum += c[i];
    printf("final result: %f \n", sum/n);

    // Release host memory
    free(a);
    free(b);
    free(c);
    return 0;
}
```

- Remove `__global__` and replace device code with `#pragma acc kernels copyin(...) copyout(...)`
- Allocate host vector memory using `malloc()`
- Initialise vectors on host (`for` loop)
- Call `vecaddgpu()` function normally to run on GPU
- Free host memory using `free()`
Vector Addition Host Code (OpenACC)

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// CUDA kernel. Each thread takes care of one element of c
void vecaddgpu(float *a, float *b, float *restrict c, int n)
{
    #pragma acc kernels copyin(a[0:n],b[0:n])
    #pragma acc kernels copyout(c[0:n])
    for (int i = 0; i < n; ++i)
        c[i] = a[i] + b[i];
}

int main(int argc, char* argv[]) {
    // Size of vectors
    int n = 100000;
    float *a;
    float *b;
    float *c;
    // Allocate memory for each vector on host
    size_t bytes = n*sizeof(float);
    a = (float*)malloc(bytes);
    b = (float*)malloc(bytes);
    c = (float*)malloc(bytes);
    // Initialize vectors on host
    int i;
    for( i = 0; i < n; i++ ) {
        a[i] = sinf(i)*sinf(i);
        b[i] = cosf(i)*cosf(i);
    }
    // Execute the kernel
    vecaddgpu(a, b, c, n);
    // Sum up vector c and print result divided by n, this should equal 1 within error
    float sum = 0;
    for( i = 0; i < n; i++ )
        sum += c[i];
    printf("final result: %f\n", sum/n);
    // Release host memory
    free(a);
    free(b);
    free(c);
    return 0;
}
```

- **Remove** `__global__` and replace device code with `#pragma acc kernels copyin(...) copyout(...)`
- **Allocate host vector memory using** `malloc()`
  - Device memory allocated inside the `copyin()` clause
- **Initialise vectors on host** (`for` loop)
  - Values copied to device using the `copyin()` clause
- **Call** `vecaddgpu` function normally to run on GPU
  - Vectors copied back to host using the `copyout()` clause
- **Free host memory using** `free()`
  - Device memory is automatically freed
OpenACC compliant compilers

- OpenACC requires a compliant compiler that understands OpenACC directives.
- GCC is NOT one of them (yet...)

The examples used the **PGI Accelerator** that is part of a software toolkit called **PGI Workstation** by **The Portland Group**.

- OpenACC directives in C and Fortran: target GPU + multicore CPU
- Also compiles “CUDA Fortran”
- Other advanced compiler optimisation routines: often 2x over gcc or VS
Compile and run

Compile your code by entering the following at the command line.

```
pgcc -acc -ta=nvidia,host -Minfo vecaddgpu.c
```

- **pgcc**: Command to invoke the C Compiler.
- **-acc**: Flag to enable the OpenACC `#pragma`'s and includes the OpenACC runtime library (i.e. `#include "openacc.h"`).
- **-ta=nvidia,host**: Flag to set the NVIDIA GPU or CPU as the target accelerator device. **Falls back to CPU if no compatible GPU at runtime!**
- **-Minfo**: Flag that displays compile-time optimization listings.
pgcc -acc -ta=nvidia -Minfo vecaddgpu.c

vecaddgpu:

5, Generating copyout(c[:n])
Generating copyin(a[:n])
Generating copyin(b[:n])
Generating compute capability 1.0 binary
Generating compute capability 2.0 binary

6, Loop is parallelizable
Accelerator kernel generated

6, #pragma acc loop gang, vector(256) /* blockIdx.x threadIdx.x */

CC 1.0 : 5 registers; 60 shared, 4 constant, 0 local memory bytes; 100% occupancy
CC 2.0 : 8 registers; 8 shared, 68 constant, 0 local memory bytes; 100% occupancy
What are gangs and vectors?

NVIDIA CUDA

Thread Block

OpenACC

Gang

Vector

Equivalent

Thread

The OpenACC execution model has three levels:

- Gang - Thread block
- Worker - Warp
- Vector - Threads in a Warp

The OpenACC compiler directives can automatically setup gangs, workers & vectors when running your code.

Advantage: Less coding effort
Disadvantage: Potentially lower speed-ups
Controlling Gangs and Vectors Inside a ‘kernels’ directive

To improve speed-ups, it is possible to control block and thread configurations in OpenACC by using the gang and vector clauses.

Example:

```c
#pragma acc kernels loop gang(100) vector(128)
for (i = 0; i < N; ++i) { ... }
```

Translation: Use a up to 100 gangs, and up to a vector length of 128 within each gang.
Controlling Gangs and Vectors
Inside a ‘parallel’ directive

To improve speed-ups, it is possible to control block and thread configurations in OpenACC by using the gang and vector clauses.

Example:

```plaintext
#pragma acc parallel num_gangs(100) vector_length(128)
#pragma acc loop
for (i = 0; i < N; ++i) { ... }
```

**Translation:**

1st pragma: Use 100 gangs, each with vector length 128.
2nd pragma: Share the work in the loop across workers
OpenACC Process Flow

Vector 0
- \( c[0] = a[0] + b[0] \)

Vector 1
- \( c[128] = a[128] + b[128] \)
- \( c[129] = a[129] + b[129] \)
- \( c[130] = a[130] + b[130] \)
- \( c[131] = a[131] + b[131] \)

Vector 2
- \( c[256] = a[256] + b[256] \)
- \( c[257] = a[257] + b[257] \)
- \( c[258] = a[258] + b[258] \)
- \( c[259] = a[259] + b[259] \)

Vector 3
- \( c[384] = a[384] + b[384] \)
- \( c[385] = a[385] + b[385] \)
- \( c[386] = a[386] + b[386] \)
- \( c[387] = a[387] + b[387] \)

Vector 4
- \( c[512] = a[512] + b[512] \)
- \( c[513] = a[513] + b[513] \)
- \( c[514] = a[514] + b[514] \)
- \( c[515] = a[515] + b[515] \)

#pragma acc kernels vector(128)
for (i = 0; i < N; i++)
  \( c[i] = a[i] + b[i] \)

OpenACC generates equivalent kernel (NB: not codegen)
Useful environment variables

C:\Working_Dir\vecaddgpu.exe

11: region entered 1 time
   time(us): total=404,000 init=45,000 region=359,000
   kernels=47,555 data=300,934
   w/o init: total=359,000 max=359,000 min=359,000 avg=359,000

13: kernel launched 1 times
   grid: [65535] block: [256]
   time(us): total=47,555 max=47,555 min=47,555 avg=47,555

PGI_ACC_TIME=1
PGI_ACC_NOTIFY=1
## Vector addition results

<table>
<thead>
<tr>
<th>n</th>
<th>Total (µs)</th>
<th>Init (µs)</th>
<th>region (µs)</th>
<th>kernels (µs)</th>
<th>data (µs)</th>
<th>CPU (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>48000</td>
<td>45200</td>
<td>2800</td>
<td>7</td>
<td>225.6</td>
<td>0</td>
</tr>
<tr>
<td>10000</td>
<td>46800</td>
<td>43800</td>
<td>3000</td>
<td>10.4</td>
<td>303.6</td>
<td>0</td>
</tr>
<tr>
<td>100000</td>
<td>49000</td>
<td>45400</td>
<td>3600</td>
<td>54.4</td>
<td>994.6</td>
<td>0</td>
</tr>
<tr>
<td>1000000</td>
<td>53400</td>
<td>46000</td>
<td>7400</td>
<td>480.6</td>
<td>4298.6</td>
<td>1400</td>
</tr>
<tr>
<td>10000000</td>
<td>82600</td>
<td>43200</td>
<td>39400</td>
<td>4707.6</td>
<td>31800</td>
<td>16200</td>
</tr>
<tr>
<td>100000000</td>
<td>403800</td>
<td>46000</td>
<td>357800</td>
<td>47562.4</td>
<td>299881.6</td>
<td>160400</td>
</tr>
</tbody>
</table>

- **GPU initialisation** is fixed; therefore it dominates for small $n$.
- **Data** transfer dominates for large $n$. 
Vector addition speedups

<table>
<thead>
<tr>
<th>n</th>
<th>GPU Speedup (kernel)</th>
<th>GPU Speedup (kernel+data)</th>
<th>Data(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
<td>0</td>
<td>0.970</td>
</tr>
<tr>
<td>10000</td>
<td>0</td>
<td>0</td>
<td>0.967</td>
</tr>
<tr>
<td>100000</td>
<td>0</td>
<td>0</td>
<td>0.948</td>
</tr>
<tr>
<td>1000000</td>
<td>2.913</td>
<td>0.293</td>
<td>0.899</td>
</tr>
<tr>
<td>10000000</td>
<td>3.441</td>
<td>0.444</td>
<td>0.871</td>
</tr>
<tr>
<td>100000000</td>
<td>3.372</td>
<td>0.462</td>
<td>0.863</td>
</tr>
</tbody>
</table>

- A 3x speedup achieved without data transfer.
- **Worse** performance when data transfer times are considered.
- Need to have **higher compute intensity** to offset data transfer overheads.
  - Each thread needs to do more work to justify the data transfer overhead!
In summary

- OpenACC is easy, powerful and portable.
- Can target Nvidia GPUs and multicore CPUs.
  - Use, `-ta=nvidia,host` flag at the command line.
- Can use the `kernels` directive to automatically parallelise code regions or the `parallel` directive or more fine-grained control.
- Currently all function calls must be `in-lineable`.
  - Changing in OpenACC 2.0.
- Important: Find where bottlenecks in your code are.
  - Use profiling tools (PGPROF, gprof, etc...)
  - Ask for help 😊
Thank You