ADVANCED OPENACC PROGRAMMING

DR. CHRISTOPH ANGERER, NVIDIA

*) THANKS TO JEFF LARKIN, NVIDIA, FOR THE SLIDES
AGENDA

- Optimizing OpenACC Loops
- Routines
- Update Directive
- Asynchronous Programming
- Multi-GPU Programming
- OpenACC Interoperability
- Atomic Directive
- Misc. Advice & Techniques
- Q&A
Identify Available Parallelism

Parallelize Loops with OpenACC

Optimize Loop Performance

Optimize Data Locality
SPARSE MATRIX/VECTOR PRODUCT

- Performs Mat/Vec product of sparse matrix
- Matrices are stored in a row-compressed format
- Parallelism per-row will vary, but is generally not very large

```fortran
99      do i=1,a*num_rows
100         tmpsum = 0.0d0
101         row_start = arow_offsets(i)
102         row_end   = arow_offsets(i+1)-1
103         do j=row_start,row_end
104             acol = acols(j)
105             acoef = acoefs(j)
106             xcoef = x(acol)
107             tmpsum = tmpsum + acoef*xcoef
108       enddo
109      y(i) = tmpsum
110      enddo
```
PARALLELIZED SPMV

Data already on device
- Compiler has vectorized the loop at 113 and selected a vector length of 256
- Total application speed-up (including other accelerated routines): 1.08X
OPENACC: 3 LEVELS OF PARALLELISM

- **Vector** threads work in lockstep (SIMD/SIMT parallelism)
- **Workers** compute a vector
- **Gangs** have 1 or more workers and share resources (such as cache, the streaming multiprocessor, etc.)
- Multiple gangs work independently of each other
OPENACC GANG, WORKER, VECTOR CLAUSES

- **gang**, **worker**, and **vector** can be added to a loop clause
- A parallel region can only specify one of each gang, worker, vector
- Control the size using the following clauses on the parallel region
  - `num_gangs(n)`, `num_workers(n)`, `vector_length(n)`

```
#pragma acc kernels loop gang
for (int i = 0; i < n; ++i)
  #pragma acc loop vector(128)
  for (int j = 0; j < n; ++j)
  ...

#pragma acc parallel vector_length(128)
#pragma acc loop gang
for (int i = 0; i < n; ++i)
  #pragma acc loop vector
  for (int j = 0; j < n; ++j)
  ...
```
OPTIMIZED SPMV VECTOR LENGTH

106  !$acc parallel loop present(arow_offsets,acols,acoefs) &
107  !$acc& private(row_start,row_end,acol,acoef,xcoef) &
108  !$acc& vector_length(32)
109  do i=1,a*num_rows
110   tmpsum = 0.0d0
111   row_start = arow_offsets(i)
112   row_end = arow_offsets(i+1)-1
113   !$acc loop vector reduction(+:tmpsum)
114  do j=row_start,row_end
115     acol = acols(j)
116     acoef = acoefs(j)
117     xcoef = x(acol)
118     tmpsum = tmpsum + acoef*xcoef
119  enddo
120  y(i) = tmpsum
121  enddo
**PERFORMANCE LIMITER: OCCUPANCY**

We need more threads!

---

<table>
<thead>
<tr>
<th>Variable</th>
<th>Achieved</th>
<th>Theoretical</th>
<th>Device Limit</th>
<th>Grid Size</th>
<th>65535 x 1</th>
<th>65535 x 1</th>
<th>Blocks x 32</th>
<th>Block Size x 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Occupancy Per SM</td>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>25% 100%</td>
<td></td>
</tr>
<tr>
<td>Active Blocks</td>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Active Warps</td>
<td>15.98</td>
<td>16</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Active Threads</td>
<td>512</td>
<td>2048</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Occupancy</td>
<td>25%</td>
<td>25%</td>
<td>100%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warps</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threads/Block</td>
<td>32</td>
<td>1024</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warps/Block</td>
<td>1</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block Limit</td>
<td>64</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
INCREASED PARALLELISM WITH WORKERS

```fortran
106  !$acc parallel loop present(arow_offsets,acols,acoefs) &
107  !$acc& private(row_start,row_end,acol,acoef,xcoef) &
108  !$acc& gang worker vector_length(32) num_workers(32)
109  do i=1,a$num_rows
110     tmpsum = 0.0d0
111     row_start = arow_offsets(i)
112     row_end   = arow_offsets(i+1)-1
113     !$acc loop vector reduction(+:tmpsum)
114     do j=row_start,row_end
115         acol = acols(j)
116         acoef = acoefs(j)
117         xcoef = x(acol)
118         tmpsum = tmpsum + acoef*xcoef
119     enddo
120     y(i) = tmpsum
121  enddo
```

![Graph showing speed-up vs number of workers with 6X to Original highlighted]
PERFORMANCE LIMITER: COMPUTE

Now we’re compute bound.
PERFORMANCE LIMITER: PARALLELISM

Really, we’re limited by parallelism per-row.
SPEED-UP STEP BY STEP

<table>
<thead>
<tr>
<th>Identify Parallelism</th>
<th>Parallelize</th>
<th>Optimize Data Locality</th>
<th>Optimize Loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00X</td>
<td>1.00X</td>
<td>2.00X</td>
<td>3.00X</td>
</tr>
<tr>
<td>4.00X</td>
<td>5.00X</td>
<td>6.00X</td>
<td>7.00X</td>
</tr>
</tbody>
</table>

Speed-up

© NVIDIA Corporation 2015
**OPENACC COLLAPSE CLAUSE**

`collapse(n):` Transform the following $n$ tightly nested loops into one, flattened loop.

- Useful when individual loops lack sufficient parallelism or more than 3 loops are nested (gang/worker/vector)

```c
#pragma acc parallel
#pragma acc loop collapse(2)
for(int i=0; i<N; i++)
  for(int j=0; j<N; j++)
...
```

```c
#pragma acc parallel
#pragma acc loop
for(int ij=0; ij<N*N; ij++)
  ...
```

**Loops must be tightly nested**
NEW CASE STUDY: MANDELBROT SET

- Application generates the image to the right.
- Each pixel in the image can be independently calculated.
- Skills Used:
  - Parallel Loop
  - Data Region
  - Update Directive
  - Asynchronous Pipelining
MANDELBROT CODE

// Calculate value for a pixel
unsigned char mandelbrot(int Px, int Py) {
    double x0=xmin+Px*dx;    double y0=ymin+Py*dy;
    double x=0.0;    double y=0.0;
    for(int i=0; x*x+y*y<4.0 && i<MAX_ITERS; i++) {
        double xtemp=x*x-y*y+x0;
        y=2*x*y+y0;
        x=xtemp;
    }
    return (double)MAX_COLOR*i/MAX_ITERS;
}

// Used in main()
for(int y=0; y<HEIGHT; y++) {
    for(int x=0; x<WIDTH; x++) {
        image[y*WIDTH+x]=mandelbrot(x,y);
    }
}

The mandelbrot() function calculates the color for each pixel.

Within main() there is a doubly-nested loop that calculates each pixel independently.
OPENACC ROUTINE DIRECTIVE

Specifies that the compiler should generate a device copy of the function/subroutine and what type of parallelism the routine contains.

Clauses:

- **gang/worker/vector/seq**
  - Specifies the level of parallelism contained in the routine.

- **bind**
  - Specifies an optional name for the routine, also supplied at call-site

- **no_host**
  - The routine will only be used on the device

- **device_type**
  - Specialize this routine for a particular device type.
MANDELBROT: ROUTINE DIRECTIVE

// mandelbrot.h
#pragma acc routine seq
unsigned char mandelbrot(int Px, int Py);

// Used in main()
#pragma acc parallel loop
for(int y=0;y<HEIGHT;y++) {
    for(int x=0;x<WIDTH;x++) {
        image[y*WIDTH+x]=mandelbrot(x,y);
    }
}

- At function source:
  - Function needs to be built for the GPU.
  - It will be called by each thread (sequentially)
- At call the compiler needs to know:
  - Function will be available on the GPU
  - It is a sequential routine
OPENACC ROUTINE: FORTRAN

The **routine** directive may appear in a Fortran function or subroutine definition, or in an interface block.

The save attribute is not supported.

Nested acc routines require the routine directive within each nested routine.

```fortran
module mandelbrot_mod
  implicit none
  integer, parameter :: HEIGHT=16384
  integer, parameter :: WIDTH=16384
  integer, parameter :: MAXCOLORS = 255
contains
    real(8) function mandlebrot(px,py)
      implicit none
      !$acc routine(mandlebrot) seq
      ... 
      end function mandlebrot
end module mandelbrot_mod
```
BASELINE PROFILE

Roughly 25% of our time is spent copying, none of it is overlapped.

We’re still much faster than the CPU because there’s a lot of work.
PIPPINING DATA TRANSFERS

Two Independent Operations Serialized

Overlapping Copying and Computation

H2D | kernel | D2H | H2D | kernel | D2H

NOTE: In real applications, your boxes will not be so evenly sized.
PIPELINING MANDELBROT SET

- We only have 1 kernel, so there’s nothing to overlap.
- Since each pixel is independent, computation can be broken up.
- Steps
  1. Break up computation into blocks along rows.
  2. Break up copies according to blocks.
  3. Make both computation and copies asynchronous.
STEP 1: BLOCKING COMPUTATION

Add a loop over blocks
Modify the existing row loop to only work within blocks
Add data region around blocking loop to leave data local to the device.
Check for correct results.

NOTE: We don’t need to copy in the array, so make it an explicit copyout.

numblocks = ( argc > 1 ) ? atoi(argv[1]) : 8;
blocksize = HEIGHT / numblocks;
printf("numblocks: %d, blocksize: %d\n",
numblocks, blocksize);

#pragma acc data copyout(image[:bytes])
for(int block=0; block < numblocks; block++)
{
   int ystart = block * blocksize;
   int yend = ystart + blocksize;
   #pragma acc parallel loop
   for(int y=ystart;y<yend;y++) {
      for(int x=0;x<WIDTH;x++) {
         image[y*WIDTH+x]=mandelbrot(x,y);
      }
   }
}
Now we have 8 kernel launches and no longer copy data to the device, but the execution time has remained the same.
#pragma acc data copyout(image[:bytes])
for(int block=0; block < numblocks; block++)
{
    int ystart = block * blocksize;
    int yend = ystart + blocksize;
    #pragma acc parallel loop
    for(int y=ystart;y<yend;y++) {
        for(int x=0;x<WIDTH;x++) {
            image[y*WIDTH+x]=mandelbrot(x,y);
        }
    }
}

Data is shared within this region.
OPENACC UPDATE DIRECTIVE

Programmer specifies an array (or part of an array) that should be refreshed within a data region.

```c
do_something_on_device()

!$acc update self(a)
```

```c
do_something_on_host()

!$acc update device(a)
```

Copy “a” from GPU to CPU

Copy “a” from CPU to GPU
STEP 2: COPY BY BLOCK

change the data region to only create the array on the GPU

use an update directive to copy individual blocks back to the host when complete

check for correct results.

```c
#pragma acc data create(image[:bytes])
for(int block=0; block < numblocks; block++)
{
    int ystart = block * blocksize;
    int yend = ystart + blocksize;
    #pragma acc parallel loop
    for(int y=ystart;y<yend;y++) {
        for(int x=0;x<WIDTH;x++) {
            image[y*WIDTH+x]=mandelbrot(x,y);
        }
    }
    #pragma acc update
    self(image[ystart*WIDTH:WIDTH*blocksize])
}
```
TIMELINE: UPDATING BY BLOCKS

We’re now updating between blocks, but not overlapping.
ASYNCHRONOUS PROGRAMMING
OPENACC ASYNC AND WAIT

async(n): launches work asynchronously in queue n
wait(n): blocks host until all operations in queue n have completed

Can significantly reduce launch latency, enables pipelining and concurrent operations

```c
#pragma acc parallel loop async(1)
...
#pragma acc parallel loop async(1)
for(int i=0; i<N; i++)
  ...
#pragma acc wait(1)
for(int i=0; i<N; i++)
```

If n is not specified, async will go into a default queue and wait will wait all previously queued work.
STEP 3: GO ASYNCHRONOUS

31  #pragma acc data create(image[:bytes])
32  for(int block=0; block < numblocks; block++)
33  {
34    int ystart = block * blocksize;
35    int yend   = ystart + blocksize;
36    #pragma acc parallel loop async(block)
37    for(int y=ystart;y<yend;y++) {
38      for(int x=0;x<WIDTH;x++) {
39        image[y*WIDTH+x]=mandelbrot(x,y);
40      }
41    }
42  #pragma acc update
43  self(image[ystart*WIDTH:WIDTH*blocksize])
44  async(block)
45  }
46  #pragma acc wait

- Make each parallel region asynchronous by placing in different queues.
- Make each update asynchronous by placing in same stream as the parallel region on which it depends.
- Synchronize for all to complete.
- Check for correct results.
Notice the kernel launches seem to take differing amounts of time. What if we tried smaller blocks?
VARYING THE NUMBER OF BLOCKS

Time (s)

2 4 8 16 32 64 128 256 512 1024
Because of the inherent load imbalance, CPU threads do really poorly here.

1. Parallelized
2. Blocked
3. Update Added
4. Asynchronous
ASYNCHRONOUS TIPS

- Reuse streams, they’re expensive to create
  - Pre-create them
  - Consider `async(block%2)` to re-use just 2 streams
- Don’t forget to `wait`
- Test with 1 stream first
MULTI-GPU PROGRAMMING
MULTI-GPU OPENACC

`acc_set_device_num(number, device_type)`
- Selects the device to use for all regions that follow

`acc_get_num_devices(device_type)`
- Queries how many devices are available of a given type

- Most often, one will set a device number once per CPU thread
MULTI-GPU MANDELBROT

for (int gpu=0; gpu < 2 ; gpu++)
{
    acc_set_device_num(gpu,acc_device_nvidia);
    #pragma acc enter data create(image[:bytes])
}

for(int block=0; block < numblocks; block++)
{
    int ystart = block * blocksize;
    int yend   = ystart + blocksize;
    acc_set_device_num(block%2,acc_device_nvidia);
    #pragma acc parallel loop async(block)
    for(int y=ystart;y<yend;y++) {
        for(int x=0;x<WIDTH;x++) {
            image[y*WIDTH+x]=mandelbrot(x,y);
        }
    }
    #pragma acc update self(image[ystart*WIDTH:WIDTH*blocksize]) async(block)
}

for (int gpu=0; gpu < 2 ; gpu++)
{
    acc_set_device_num(gpu,acc_device_nvidia);
    #pragma acc wait
    #pragma acc exit data delete(image)
}
MULTI-GPU MANDELBROT PROFILE
OPENACC INTEROPERABILITY
OPENACC INTEROPERABILITY

OpenACC plays well with others.

- Add CUDA or accelerated libraries to an OpenACC application
- Add OpenACC to an existing accelerated application
- Share data between OpenACC and CUDA
OPENACC & CUDA STREAMS

OpenACC suggests two functions for interoperating with CUDA streams:

- `void* acc_get_cuda_stream(int async);`
- `int acc_set_cuda_stream(int async, void* stream);`
OPENACC HOST_DATA DIRECTIVE

Exposes the device address of particular objects to the host code.

```c
#pragma acc data copy(x,y)
{
// x and y are host pointers
#pragma acc host_data use_device(x,y)
{
    // x and y are device pointers
}
// x and y are host pointers
}
```

X and Y are device pointers here
HOST_DATA EXAMPLE

OpenACC Main

```fortran
program main
    integer, parameter :: N = 2**20
    real, dimension(N) :: X, Y
    real :: A = 2.0

    !$acc data
    ! Initialize X and Y
    ...
    !$acc host_data use_device(x,y)
    call saxpy(n, a, x, y)
    !$acc end host_data
    !$acc end data

end program
```

CUDA C Kernel & Wrapper

```c
__global__
void saxpy_kernel(int n, float a,
                  float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

void saxpy(int n, float a, float *dx, float *dy)
{
    // Launch CUDA Kernel
    saxpy_kernel<<<4096,256>>>(N, 2.0, dx, dy);
}
```

- It's possible to interoperate from C/C++ or Fortran.
- OpenACC manages the data and passes device pointers to CUDA.

- CUDA kernel launch wrapped in function expecting device arrays.
- Kernel is launch with arrays passed from OpenACC in main.
OpenACC can interface with existing GPU-optimized libraries (from C/C++ or Fortran).

This includes...
- CUBLAS
- Libsci_acc
- CUFFT
- MAGMA
- CULA
- Thrust
- ...

```c
int N = 1<<20;
float *x, *y
// Allocate & Initialize X & Y
...
cublasInit();
#pragma acc data copyin(x[0:N]) copy(y[0:N])
{
    #pragma acc host_data use_device(x,y)
    {
        // Perform SAXPY on 1M elements
        cublasSaxpy(N, 2.0, x, 1, y, 1);
    }
}
cublasShutdown();
```
OPENACC DEVICEPTR

The `deviceptr` clause informs the compiler that an object is already on the device, so no translation is necessary.

- Valid for `parallel`, `kernels`, and `data`

```c
cudaMallocManaged((void**)&x,(size_t)n*sizeof(float));
cudaMallocManaged((void**)&y,(size_t)n*sizeof(float));

#pragma acc parallel loop deviceptr(x,y)
for(int i=0; i<n ; i++)
{
    y(i) = a*x(i)+y(i)
}
```

Do not translate `x` and `y`, they are already on the device.
By passing a device pointer to an OpenACC region, it’s possible to add OpenACC to an existing CUDA code. Memory is managed via standard CUDA calls.
Thrust (thrust.github.io) is a STL-like library for C++ on accelerators.

- High-level interface
- Host/Device container classes
- Common parallel algorithms

It’s possible to cast Thrust vectors to device pointers for use with OpenACC.

```cpp
void saxpy(int n, float a, float * restrict x, float * restrict y)
{
    #pragma acc kernels deviceptr(x[0:n], y[0:n])
    {
        for(int i=0; i<n; i++)
        {
            y[i] += 2.0*x[i];
        }
    }
}
```

```cpp
int N = 1<<20;
thrust::host_vector<float> x(N), y(N);
for(int i=0; i<N; i++)
{
    x[i] = 1.0f;
    y[i] = 0.0f;
}

// Copy to Device
thrust::device_vector<float> d_x = x;
thrust::device_vector<float> d_y = y;
saxpy(N,2.0, d_x.data().get(),
    d_y.data().get());

// Copy back to host
y = d_y;
```
Even CUDA __device__ functions can be called from OpenACC if declared with acc routine.
OPENACC ACC_MAP_DATA FUNCTION

The **acc_map_data** (acc_unmap_data) maps (unmaps) an existing device allocation to an OpenACC variable.

```c
cudaMalloc((void**)&x_d,(size_t)n*sizeof(float));
acc_map_data(x, x_d, n*sizeof(float));
cudaMalloc((void**)&y_d,(size_t)n*sizeof(float));
acc_map_data(y, y_d, n*sizeof(float));

#pragma acc parallel loop
for(int i=0; i<n ; i++)
{
  y(i) = a*x(i)+y(i)
}
```

Allocate device arrays with CUDA and map to OpenACC

Here x and y will reuse the memory of x_d and y_d
ATOMIC DIRECTIVE
OPENACC ATOMIC DIRECTIVE

**atomic**: subsequent block of code is performed atomically with respect to other threads on the accelerator

**Clauses**: read, write, update, capture

```c
#pragma acc parallel loop
for(int i=0; i<N; i++) {
    #pragma acc atomic update
    a[i%100]++;
}
```
OPENACC ATOMIC: HISTOGRAM

19  #pragma acc data copyin(a[0:N]) copyout(h[0:HN])
20   for(int it=0;it<ITERS;it++)
21   {
22     #pragma acc parallel loop
23       for(int i=0;i<HN;i++)
24         h[i]=0;
25
26     #pragma acc parallel loop
27       for(int i=0;i<N;i++) {
28         #pragma acc atomic
29         h[a[i]]+=1;
30       }
31   }
MISC. ADVICE AND TECHNIQUES
WRITE PARALLELIZABLE LOOPS

Use countable loops
C99: while->for
Fortran: while->do

Avoid pointer arithmetic (use array syntax)

Write rectangular loops (compiler cannot parallelize triangular loops)

```cpp
bool found=false;
while(!found && i<N) {
    if(a[i]==val) {
        found=true
        loc=i;
    }
    i++;
}
```

```cpp
bool found=false;
for(int i=0;i<N;i++) {
    if(a[i]==val) {
        found=true
        loc=i;
    }
}
```

```cpp
for(int i=0;i<N;i++) {
    for(int j=i;j<N;j++) {
        sum+=A[i][j];
    }
}
```

```cpp
for(int i=0;i<N;i++) {
    for(int j=0;j<N;j++) {
        if(j>=i)
            sum+=A[i][j];
    }
}
```
ALIASING CAN PREVENT PARALLELIZATION

23, Loop is parallelizable
   Accelerator kernel generated
   23, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */

25, Complex loop carried dependence of 'b->' prevents parallelization
   Loop carried dependence of 'a->' prevents parallelization
   Loop carried backward dependence of 'a->' prevents vectorization
   Accelerator scalar kernel generated

27, Complex loop carried dependence of 'a->' prevents parallelization
   Loop carried dependence of 'b->' prevents parallelization
   Loop carried backward dependence of 'b->' prevents vectorization
   Accelerator scalar kernel generated
C99: RESTRICT KEYWORD

- Declaration of intent given by the programmer to the compiler
  Applied to a pointer, e.g.
  \[
  \text{float *restrict } \text{ptr}
  \]
  Meaning: “for the lifetime of \text{ptr}, only it or a value directly derived from it (such as \text{ptr} + 1) will be used to access the object to which it points”*

- Parallelizing compilers often require \text{restrict} to determine independence
  - Otherwise the compiler can’t parallelize loops that access \text{ptr}
  - Note: if programmer violates the declaration, behavior is undefined

\[
\text{float restrict *ptr} \\
\text{float *restrict } \text{ptr}
\]

http://en.wikipedia.org/wiki/Restrict
INLINING

- When possible aggressively inline functions/routines
  - This is especially important for inner loop calculations
  - Inlined routines frequently perform better than acc routines because the compiler has more information.

```c
#pragma acc routine seq
inline
int IDX(int row, int col, int LDA) {
    return row*LDA+col;
}
```
KERNEL FUSION

- Kernel calls are expensive
  - Each call can take over 10us in order to launch
  - It is often a good idea to combine loops of same trip counts containing very few lines of code
- Kernel Fusion (i.e. Loop fusion)
  - Join nearby kernels into a single kernel

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

- Loops that are exceptionally long may result in kernels that are resource-bound, resulting in low GPU occupancy.
- This is particularly true for outer parallel loops containing nested loops
- Caution: This may introduce temporaries.

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

- **Coalesced** access:
  - A group of 32 contiguous threads ("warp") accessing adjacent words
  - Few transactions and high utilization

- **Uncoalesced** access:
  - A warp of 32 threads accessing scattered words
  - Many transactions and low utilization

- For best performance the **vector loop** should access memory **contiguously** (stride-1)
COMPLEX DATA LAYOUTS

- OpenACC works best with flat arrays
- Some compilers handle complex types (structs, classes, derived types) better than others
  - Doesn’t always work, particularly if members are dynamically allocated
  - Work around: Use local pointers to struct members (C99 & Fortran)

```c
#pragma acc parallel loop \
    copy(a, a.data[0:a.N])
for(i=0;i<a.N;i++)
a.data[i]=0;
```

```c
int N=a.N;
float *data=a.data;
#pragma acc parallel loop \
    copy(data[0:N])
for(i=0;i<N;i++)
data[i]=0;
```

May work

Generally Works
OPENACC INDEPENDENT CLAUSE

Specifies that loop iterations are data independent. This overrides any compiler dependency analysis. This is implied for parallel loop.

```c
#pragma acc kernels
{
#pragma acc loop independent
for(int i=0; i<N; i++)
{
    a[i] = 0.0;
    b[i] = 1.0;
    c[i] = 2.0;
}
#pragma acc loop independent
for(int i=0; i<N; i++)
{
    a(i) = b(i) + c(i)
}
}
```

Informs the compiler that both loops are safe to parallelize so it will generate both kernels.
OPENACC DEBUGGING

- Most OpenACC directives accept an if(condition) clause

```c
#pragma acc update self(A) if(debug)
#pragma acc parallel loop if(!debug)
[…]
#pragma acc update device(A) if(debug)
```

- Use default(none) to force explicit data directives

```c
#pragma acc data copy(...) create(...) default(none)
```
QUESTIONS & DISCUSSION