Lecture 22: Manycore GPU Architectures and Programming, Part 4
-- Introducing OpenMP and OpenACC for Accelerators

Concurrent and Multicore Programming

Department of Computer Science and Engineering
Yonghong Yan
yan@oakland.edu
www.secs.oakland.edu/~yan
Manycore GPU Architectures and Programming: Outline

• Introduction
  – GPU architectures, GPGPUs, and CUDA
• GPU Execution model
• CUDA Programming model
• Working with Memory in CUDA
  – Global memory, shared and constant memory
• Streams and concurrency
• CUDA instruction intrinsic and library
• Performance, profiling, debugging, and error handling

Directive-based high-level programming model
  – OpenMP and OpenACC
OpenMP 4.0 for Accelerators

- **Device:** a logical execution engine
  - Host device: where OpenMP program begins, one only
  - Target devices: **1 or more** accelerators

- **Memory model**
  - Host data environment: one
  - Device data environment: one or more
  - Allow shared host and device memory

- **Execution model:** Host-centric
  - Host device: “offloads” code regions and data to accelerators/target devices
  - Target Devices: still fork-join model
  - Host waits until devices finish
  - Host executes device regions if no accelerators are available /supported
Computation and data offloading for accelerators (2.9)

- **#pragma omp target** `device(id) map() if()`
  - `target`: create a data environment and offload computation on the device
  - `device (int_exp)`: specify a target device
  - `map(to|from|tofrom|alloc:var_list)`: data mapping between the current data environment and a device data environment

- **#pragma target data** `device (id) map() if()`
  - Create a device data environment: to be reused/inherited

![Diagram of data flow and thread mapping]

- **omp target**
- **omp parallel**
- **CPU thread**
- **Accelerator threads**
- **Main Memory**
- **Application data**
- **Copy in remote data**
- **Copy out remote data**
- **Tasks offloaded to accelerator**
- **acc. cores**
- **Application data**

[Diagram showing data flow and thread mapping between CPU and accelerator cores]
target and map examples

```c
void vec_mult(int N)
{
    int i;
    float p[N], v1[N], v2[N];
    init(v1, v2, N);
    #pragma omp target map(to: v1, v2) map(from: p)
    #pragma omp parallel for
    for (i=0; i<N; i++)
        p[i] = v1[i] * v2[i];
    output(p, N);
}

void vec_mult(float *p, float *v1, float *v2, int N)
{
    int i;
    init(v1, v2, N);
    #pragma omp target map(to: v1[0:N], v2[:N]) map(from: p[0:N])
    #pragma omp parallel for
    for (i=0; i<N; i++)
        p[i] = v1[i] * v2[i];
    output(p, N);
}
```
Accelerator: explicit data mapping

- Relatively small number of truly shared memory accelerators so far
- Require the user to explicitly map data to and from the device memory
- Use array region

```c
long a = 0x858;
long b = 0;
int anArray[100]

#pragma omp target data map(to:a) \ \ 
   map(tofrom:b,anArray[0:64])
{
    /* a, b and anArray are mapped
       * to the device */

    /* work on the device */
    #pragma omp target ...
    {
    ... 
    }
}

/* b and anArray are mapped
 * back to the host */
```
void vec_mult(float *p, float *v1, float *v2, int N) {
    int i;
    init(v1, v2, N);
    #pragma omp target data map(from: p[0:N])
    {
        #pragma omp target map(to: v1[:N], v2[:N])
        #pragma omp parallel for
        for (i=0; i<N; i++)
            p[i] = v1[i] * v2[i];
        init_again(v1, v2, N);
    }
    #pragma omp target map(to: v1[:N], v2[:N])
    #pragma omp parallel for
    for (i=0; i<N; i++)
        p[i] = p[i] + (v1[i] * v2[i]);
    output(p, N);
}
Accelerator: hierarchical parallelism

- Organize massive number of threads
  - teams of threads, e.g. map to CUDA grid/block
- Distribute loops over teams

```c
#pragma omp target
#pragma omp teams num_teams(2) num_threads(8)
{
    //-- creates a “league” of teams
    //-- only local barriers permitted
    #pragma omp distribute
    for (int i=0; i<N; i++) {
    }
}
```
teams and distribute loop example

```c
float dotprod_teams(float B[], float C[], int N, int num_blocks,
int block_threads)
{
    float sum = 0;
    int i, i0;
    #pragma omp target map(to: B[0:N], C[0:N])
    #pragma omp teams num_teams(num_blocks) thread_limit(block_threads)
    reduction(+:sum)
    #pragma omp distribute
    for (i0=0; i0<N; i0 += num_blocks)
        #pragma omp parallel for reduction(+:sum)
        for (i=i0; i< min(i0+num_blocks,N); i++)
            sum += B[i] * C[i];
    return sum;
}
```

Double-nested loops are mapped to the two levels of thread hierarchy (league and team)
OpenMP 4.0

• Released July 2013
  – A document of examples is expected to release soon
• Changes from 3.1 to 4.0 (Appendix E.1):
  – Accelerator: 2.9
  – SIMD extensions: 2.8
  – Places and thread affinity: 2.5.2, 4.5
  – Taskgroup and dependent tasks: 2.12.5, 2.11
  – Error handling: 2.13
  – User-defined reductions: 2.15
  – Sequentially consistent atomics: 2.12.6
  – Fortran 2003 support
OpenACC

• OpenACC’s guiding principle is simplicity
  – Want to remove as much burden from the programmer as possible
  – No need to think about data movement, writing kernels, parallelism, etc.
  – OpenACC compilers automatically handle all of that

• In reality, it isn’t always that simple
  – Don’t expect to get massive speedups from very little work

• However, OpenACC can be an easy and straightforward programming model to start with
OpenACC

• OpenACC shares a lot of principles with OpenMP
  – Compiler `#pragma` based, and requires a compiler that supports OpenACC
  – Express the type of parallelism, let the compiler and runtime handle the rest
  – OpenACC also allows you to express data movement using compiler `#pragmas`

`#pragma acc`
OpenACC Directives

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 Parallelizes code
Works on many-core GPUs & multicore CPUs
OpenACC

• Creating parallelism in OpenACC is possible with either of the following two compute directives:

  #pragma acc kernels
  #pragma acc parallel

• kernels and parallel each have their own strengths
  – kernels is a higher abstraction with more automation
  – parallel offers more low-level control but also requires more work from the programmer
OpenACC Compute Directives

• The `kernels` directive marks a code region that the programmer wants to execute on an accelerator
  – The code region is analyzed for parallelizable loops by the compiler
  – Necessary data movement is also automatically generated

```c
#pragma acc kernels
{
    for (i = 0; i < N; i++)
        C[i] = A[i] + B[i];

    for (i = 0; i < N; i++)
        D[i] = C[i] * A[i];
}
```
OpenACC Compute Directives

• Like OpenMP, OpenACC compiler directives support clauses which can be used to modify the behavior of OpenACC

  #pragmas

  #pragma acc kernels clause1 clause2 ...

• kernels supports a number of clauses, for example:
  – if(cond) : Only run the parallel region on an accelerator if cond is true
  – async(id) : Don’t wait for the parallel code region to complete on the accelerator before returning to the host application. Instead, id can be used to check for completion.
  – wait(id) : wait for the async work associated with id to finish first
  – ...

OpenACC Compute Directives

• Take a look at the `simple-kernels.c` example

  – Compile with an OpenACC compiler, e.g. PGI:
    
    ```bash
    $ pgcc -acc simple-kernels.c -o simple-kernels
    ```

  – You may be able to add compiler-specific flags to print more diagnostic information on the accelerator code generation, e.g.:
    
    ```bash
    $ pgcc -acc simple-kernels.c -o simple-kernels -Minfo=accel
    ```

We donot have this compiler on our systems
On the other hand, the parallel compute directive offers much more control over exactly how a parallel code region is executed.

- With just kernels, we have little control over which loops are parallelized or how they are parallelized.
- Think of `#pragma acc parallel` similarly to `#pragma omp parallel`.

```bash
#pragma acc parallel
```
OpenACC Compute Directives

• **With `parallel`, all parallelism is created at the start of the parallel region and does not change until the end**
  - The execution mode of a parallel region changes depending on programmer-inserted `#pragmas`

• **`parallel` supports similar clauses to `kernels`, plus:**
  - `num_gangs(g), num_workers(w), vector_length(v)`: **Used to configure the amount of parallelism in a parallel region**
  - `reduction(op:var1, var2, ...): Perform a reduction across gangs of the provided variables using the specified operation`
  - `...`
OpenACC

- Mapping from the abstract GPU Execution Model to OpenACC concepts and terminology
  - *OpenACC Vector element* = a thread
    - The use of “vector” in OpenACC terminology emphasizes that at the lowest level, OpenACC uses vector-parallelism
  - *OpenACC Worker* = SIMT Group
    - Each worker has a vector width and can contain many vector elements
  - *OpenACC Gang* = SIMT Groups on the same SM
    - One gang per OpenACC PU
    - OpenACC supports multiple gangs executing concurrently
OpenACC

• Mapping to CUDA threading model:
  
  – Gang Parallelism: Work is run across multiple OpenACC Pus
    • CUDA Blocks
  – Worker Parallelism: Work is run across multiple workers (i.e. SIMT Groups)
    • Threads per Blocks
  – Vector Parallelism: Work is run across vector elements (i.e. threads)
    • Within Wrap
OpenACC Compute Directives

• In addition to kernels and parallel, a third OpenACC compute directive can help control parallelism (but does not actually create threads):

  #pragma acc loop

• The loop directive allows you to explicitly mark loops as parallel and control the type of parallelism used to execute them
OpenACC Compute Directives

- **Using** `#pragma acc loop gang/worker/vector` allows you to explicitly mark loops that should use gang, worker, or vector parallelism in your OpenACC application
  - Can be used inside both **parallel** and **kernels regions**

- **Using** `#pragma acc independent` allows you to explicitly mark loops as parallelizable, overriding any automatic compiler analysis
  - Compilers must naturally be conservative when auto-parallelizing, the **independent** clause allows you to use detailed knowledge of the application to give hints to the compiler
OpenACC Compute Directives

• Consider simple-parallel.c, in which the loop and parallel directives are used to implement the same computation as simple-kernels.c

```c
#pragma acc parallel
{
    #pragma acc loop
    for (i = 0; i < N; i++)
    ...

    #pragma acc loop
    for (i = 0; i < N; i++)
    ...
}
```
OpenACC Compute Directives

• As a syntactic nicety, you can combine parallel/kernels directives with loop directives:

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

#pragma acc parallel loop
for (i = 0; i < N; i++) {
    ...
}
```
OpenACC Compute Directives

• This combination has the same effect as a `loop` directive immediately following a `parallel/kernels` directive:

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

#pragma acc parallel
#pragma acc loop
for (i = 0; i < N; i++) { ... }
```
OpenACC Compute Directives

In summary, the kernels, parallel, and loop directives all offer different ways to control the OpenACC parallelism of an application:

- **kernels** is highly automated, but you rely heavily on the compiler to create an efficient parallelization strategy.
  - A short-form of parallel/loop for GPU
- **parallel** is more manual, but allows programmer knowledge about the application to improve the parallelization strategy.
  - Like OpenMP parallel
- **loop** allows you to take more manual control over both.
  - Like OpenMP worksharing
Suggested Readings

1. The sections on *Using OpenACC* and *Using OpenACC Compute Directives* in Chapter 8 of *Professional CUDA C Programming*


OpenACC Data Directives

• `#pragma acc data` can be used to explicitly perform communication between a host program and accelerators.

• The `data` clause is applied to a code region and defines the communication to be performed at the start and end of that code region.

• The `data` clause alone does nothing, but it takes clauses which define the actual transfers to be performed.
OpenACC Data Directives

- **Common clauses used with #pragma acc data:**

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>copy (list)</strong></td>
<td>Transfer all variables in <code>list</code> to the accelerator at the start of the <code>data</code> region and back to the host at the end.</td>
</tr>
<tr>
<td><strong>copyin (list)</strong></td>
<td>Transfer all variables in <code>list</code> to the accelerator at the start of the <code>data</code> region.</td>
</tr>
<tr>
<td><strong>copyout (list)</strong></td>
<td>Transfer all variables in <code>list</code> back to the host at the end of the <code>data</code> region.</td>
</tr>
<tr>
<td><strong>present_or_copy(list)</strong></td>
<td>If the variables specified in <code>list</code> are not already on the accelerator, transfer them to it at the start of the <code>data</code> region and back at the end.</td>
</tr>
<tr>
<td><strong>if (cond)</strong></td>
<td>Only perform the operations defined by this data directive if <code>cond</code> is true.</td>
</tr>
</tbody>
</table>
OpenACC Data Directives

- Consider the example in `simple-data.c`, which mirrors `simple-parallel.c` and `simple-kernels.c`:

```c
#pragma acc data copyin(A[0:N], B[0:N])
copyout(C[0:N], D[0:N])
{
#pragma acc parallel
{
#pragma acc loop
    for (i = 0; i < N; i++)
        ...
#pragma acc loop
    for (i = 0; i < N; i++)
        ...
}
}
```
OpenACC Data Directives

• OpenACC also supports:
  
  #pragma acc enter data
  #pragma acc exit data

• Rather than bracketing a code region, these #pragmas allow you to copy data to and from the accelerator at arbitrary points in time
  
  – Data transferred to an accelerator with enter data will remain there until a matching exit data is reached or until the application terminates
OpenACC Data Directives

- Finally, OpenACC also allows you to specify data movement as part of the compute directives through data clauses

```c
#pragma acc data copyin(A[0:N], B[0:N])
copyout(C[0:N], D[0:N])
{
#pragma acc parallel
    {
    #pragma acc parallel
        {
        }
    }
}

#pragma acc parallel copyin(A[0:N], B[0:N])
copyout(C[0:N], D[0:N])
```
OpenACC Data Specification

• You may have noticed that OpenACC data directives use an unusual array dimension specification, for example:

```
#pragma acc data copy(A[start:length])
```

• In some cases, data specifications may not even be necessary as the OpenACC compiler can infer the size of the array:

```
int a[5];
#pragma acc data copy(a)
{
    ...
}
```
If the compiler is unable to infer an array size, error messages like the one below will be emitted:

Example code:

```c
int *a = (int *)malloc(sizeof(int) * 5);
#pragma acc data copy(a)
{
    ...
}
```

Example error message:

```
PGCC-S-0155-Cannot determine bounds for array a
```
OpenACC Data Specification

• Instead, you must specify the full array bounds to be transferred

```c
int *a = (int *)malloc(sizeof(int) * 5);
#pragma acc data copy(a[0:5])
{
    ...
}
```

– The lower bound is inclusive and, if not explicitly set, will default to 0
– The length must be provided if it cannot be inferred
Asynchronous Work in OpenACC

• In OpenACC, the default behavior is always to block the host while executing an acc region
  – Host execution does not continue past a kernels/parallel region until all operations within it complete
  – Host execution does not enter or exit a data region until all prescribed data transfers have completed
Asynchronous Work in OpenACC

• When the host blocks, host cycles are wasted:
Asynchronous Work in OpenACC

• In many cases this default can be overridden to perform operations asynchronously
  – Asynchronously copy data to the accelerator
  – Asynchronously execute computation

• As a result, host cycles are not wasted idling while the accelerator is working
Asynchronous Work in OpenACC

- Asynchronous work is created using the `async` clause on compute and data directives, and every asynchronous task has an `id`
  - Run a kernels region asynchronously:
    ```
    #pragma acc kernels async(id)
    ```
  - Run a parallel region asynchronously:
    ```
    #pragma acc parallel async(id)
    ```
  - Perform an enter data asynchronously:
    ```
    #pragma acc enter data async(id)
    ```
  - Perform an exit data asynchronously:
    ```
    #pragma acc exit data async(id)
    ```
  - `async` is not supported on the data directive
Asynchronous Work in OpenACC

• Having asynchronous work means we also need a way to wait for it
  – Note that every async clause on the previous slide took an id
  – The asynchronous task created is uniquely identified by that id

• We can then wait on that id using either:
  – The wait clause on compute or data directives
  – The OpenACC Runtime API’s Asynchronous Control functions
Asynchronous Work in OpenACC

• Adding a `wait(id)` clause to a compute or data directive makes the associated data transfer or computation wait until the asynchronous task associated with that `id` completes.

• The OpenACC Runtime API supports explicitly waiting using:
  ```c
  void acc_wait(int id);
  void acc_wait_all();
  ```

• You can also check if asynchronous tasks have completed using:
  ```c
  int acc_async_test(int id);
  int acc_async_test_all();
  ```
Asynchronous Work in OpenACC

• Let’s take a simple code snippet as an example:

```c
#pragma acc data copyin(A[0:N])
copyout(B[0:N])
{
    #pragma acc kernels
    {
        for (i = 0; i < N; i++)
            B[i] = foo(A[i]);
    }
}
do_work_on_host(C);
```
Asynchronous Work in OpenACC

Single-threaded host

Accelerator w/ many PUs

<table>
<thead>
<tr>
<th>copyin</th>
<th>Idling</th>
<th>copyout</th>
<th>do_work_on_host</th>
</tr>
</thead>
</table>

acc kernels
Asynchronous Work in OpenACC

• Performing the transfer and compute asynchronously allows us to overlap the host and accelerator work:

```c
#pragma acc enter data async(0) copyin(A[0:N]) create(B[0:N])
#pragma acc kernels wait(0) async(1)
{
    for (i = 0; i < N; i++)
        B[i] = foo(A[i]);
}
#pragma acc exit data wait(1) async(2) copyout(B[0:N])
do_work_on_host(C);
acc_wait(2);
```
Asynchronous Work in OpenACC

Single-threaded host

Accelerator w/ many PUs

do_work_on_host

acc kernels
Reductions in OpenACC

• OpenACC supports the ability to perform automatic parallel reductions
  – The reduction clause can be added to the parallel and loop directives, but has a subtle difference in meaning on each
    
    #pragma acc parallel reduction(op:var1, var2, ...)
    #pragma acc loop reduction(op:var1, var2, ...)

  – op defines the reduction operation to perform
  – The variable list defines a set of private variables created and initialized in the subsequent compute region
Reductions in OpenACC

• When applied to a parallel region, reduction creates a private copy of each variable for each gang created for that parallel region.

• When applied to a loop directive, reduction creates a private copy of each variable for each vector element in the loop region.

• The resulting value is transferred back to the host once the current compute region completes.
OpenACC Parallel Region Optimizations

• To some extent, optimizing the parallel code regions in OpenACC is contradictory to the whole OpenACC principle
  – OpenACC wants programmers to focus on writing application logic and worry less about nitty-gritty optimization tricks
  – Often, low-level code optimizations require intimate understanding of the hardware you are running on

• In OpenACC, optimizing is more about avoiding symptomatically horrible scenarios so that the compiler has the best code to work with, rather than making very low-level optimizations
  – Memory access patterns
  – Loop scheduling
OpenACC Parallel Region Optimizations

• GPUs are optimized for aligned, coalesced memory accesses
  – Aligned: the lowest address accessed by the elements in a vector to be 32- or 128-bit aligned (depending on architecture)
  – Coalesced: neighboring vector elements access neighboring memory cells
OpenACC Parallel Region Optimizations

• Improving alignment in OpenACC is difficult because there is less visibility into how OpenACC threads are scheduled on GPU.

• Improving coalescing is also difficult, the OpenACC compiler may choose a number of different ways to schedule a loop across threads on the GPU.

• In general, try to ensure that neighboring iterations of the innermost parallel loops are referencing neighboring memory cells.
OpenACC Parallel Region Optimizations

- Vecadd example using coalescing and noncoalescing access

<table>
<thead>
<tr>
<th>CLI Flag</th>
<th>Average Compute Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without (-b) (coalescing)</td>
<td>122.02us</td>
</tr>
<tr>
<td>With (-b) (noncoalescing)</td>
<td>624.04ms</td>
</tr>
</tbody>
</table>
OpenACC Parallel Region Optimizations

- The loop directive supports three special clauses that control how loops are parallelized: gang, worker, and vector
  - The meaning of these clauses changes depending on whether they are used in a parallel or kernels region

- The gang clause:
  - In a parallel region, causes the iterations of the loop to be parallelized across gangs created by the parallel region, transitioning from gang-redundant to gang-partitioned mode.
  - In a kernels region, does the same but also allows the user to specify the number of gangs to use, using gang (ngangs)
• The **worker** clause:
  
  - **In a parallel region**, causes the iterations of the loop to be parallelized across workers created by the **parallel region**, transitioning from worker-single to worker-partitioned modes.
  
  - **In a kernels region**, does the same but also allows the user to specify the number of workers per gang, using `worker(nworkers)`
OpenACC Parallel Region Optimizations

- The `vector` clause:
  - In a parallel region, causes the iterations of the loop to be parallelized using vector/SIMD parallelism with the vector length specified by `parallel`, transitioning from vector-single to vector-partitioned modes.
  - In a kernels region, does the same but also allows the user to specify the vector length to use, using `vector(vector_length)`
OpenACC Parallel Region Optimizations

• Manipulating the *gang*, *worker*, and *vector* clauses results in different scheduling of loop iterations on the underlying hardware
  – Can result in significant performance improvement or loss

• Consider the example of loop schedule
  – The *gang* and *vector* clauses are used to change the parallelization of two nested loops in a parallel region
  – The # of gangs is set with the command-line flag `-g`, vector width is set with `-v`
OpenACC Parallel Region Optimizations

- Try playing with \(-g\) and \(-v\) to see how gang and vector affect performance
  - Options for gang and vector sizes

```c
#pragma acc parallel copyin(A[0:M * N], B[0:M * N]) copyout(C[0:M * N])
#pragma acc loop gang(gangs)
    for (int i = 0; i < M; i++) {
        #pragma acc loop vector(vector_length)
            for (int j = 0; j < N; j++) {
                ...
            }
    }
```
OpenACC Parallel Region Optimizations

Example results:

<table>
<thead>
<tr>
<th>-g (constant)</th>
<th>-v (constant)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>128</td>
<td>5.7590ms</td>
</tr>
<tr>
<td>2</td>
<td>128</td>
<td>2.8855ms</td>
</tr>
<tr>
<td>4</td>
<td>128</td>
<td>1.4478ms</td>
</tr>
<tr>
<td>8</td>
<td>128</td>
<td>730.11us</td>
</tr>
<tr>
<td>16</td>
<td>128</td>
<td>373.40us</td>
</tr>
<tr>
<td>32</td>
<td>128</td>
<td>202.89us</td>
</tr>
<tr>
<td>64</td>
<td>128</td>
<td>129.85us</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>-g (constant)</th>
<th>-v (constant)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>2</td>
<td>9.3165ms</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>2.7953ms</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>716.45us</td>
</tr>
<tr>
<td>32</td>
<td>128</td>
<td>203.02us</td>
</tr>
<tr>
<td>32</td>
<td>256</td>
<td>129.76us</td>
</tr>
<tr>
<td>32</td>
<td>512</td>
<td>125.16us</td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>124.83us</td>
</tr>
</tbody>
</table>
OpenACC Parallel Region Optimizations

• Your options for optimizing OpenACC parallel regions are fairly limited
  – The whole idea of OpenACC is that the compiler can handle that for you

• There are some things you can do to avoid poor code characteristics on the GPU that that compiler can’t optimize you out of (memory access patterns)

• There are also tunables you can tweak which may improve performance (e.g. gang, worker, vector)
The Tile Clause

• Like the **gang**, **worker**, and **vector clauses**, the **tile clause** is used to control the scheduling of loop iterations
  – **Used on loop directives only**

• It specifies how you would like loop iterations grouped across the iteration space
  – **Iteration grouping (more commonly called loop tiling)** can be beneficial for locality on both CPUs and GPUs
The Tile Clause

• Suppose you have a loop like the following:

```c
#pragma loop
for (int i = 0; i < N; i++) {
    ...
}
```

• The `tile` clause can be added like this:

```c
#pragma loop tile(8)
for (int i = 0; i < N; i++) {
    ...
}
```
The Tile Clause

- Analogous to adding a second inner loop:

```c
#pragma loop
for (int i = 0; i < N; i+=8) {
    for (int ii = 0; ii < 8; ii++) {
        ...
    }
}
```

- The same iterations are performed, but the compiler may choose to schedule them differently on hardware threads
The Cache Directive

- The *cache* directive is used to optimize memory accesses on the accelerator. It marks data which will be frequently accessed, and which therefore should be kept close in the cache hierarchy.

- The *cache* directive is applied immediately inside of a loop that is being parallelized on the accelerator:
  - Note the same data specification is used here as for data directives

```c
#pragma acc loop
for (int i = 0; i < N; i++) {
  #pragma acc cache(A[i:1])
  ...
```
The Cache Directive

- For example, suppose you have an application where every thread \( i \) accesses cells \( i-1, i, \) and \( i+1 \) in a vector \( A \).
The Cache Directive

• This results in lots of wasted memory accesses as neighboring elements in the vector reference the same cells in the array \( A \)

• Instead, we can use the cache directive to indicate to the compiler which array elements we expect to benefit from caching:

```c
#pragma acc parallel loop
for (int i = 0; i < N; i++) {
}
```

```c
#pragma acc parallel loop
for (int i = 0; i < N; i++) {
    #pragma acc cache(A[i-1:2])
}
```
The Cache Directive

- Now, the compiler will automatically cache $A[i-1]$, $A[i]$, and $A[i+1]$ and only load them from accelerator memory once.
The Cache Directive

• The cache directive requires a lot of complex code analysis from the compiler to ensure this is a safe optimization

• As a result, it is not always possible to use the cache optimization with arbitrary application code
  – Some restructuring may be necessary before the compiler is able to determine how to effectively use the cache optimization
The Cache Directive

- The cache directive can result in significant performance gains thanks to much improved data locality.

- However, for complex applications it generally requires significant code refactoring to expose the cache-ability of the code to the compiler.
  - Just like to use shared memory in CUDA.
Suggested Readings
