Harness the Power of GPUs:  
An Introduction to GPGPU Programming  
Lecture 5: Advanced OpenMP/OpenACC

Guido Juckeland  
Visiting Scholar  
Technische Universität Dresden, Germany

July 9, 2014
Locking memory

• Simple Lock routines:
  – A simple lock is available if it is unset.
    • `omp_init_lock()`, `omp_set_lock()`, `omp_unset_lock()`, `omp_test_lock()`, `omp_destroy_lock()`

• Nested Locks
  – A nested lock is available if it is unset or if it is set but owned by the thread executing the nested lock function
    • `omp_init_nest_lock()`, `omp_set_nest_lock()`, `omp_unset_nest_lock()`, `omp_test_nest_lock()`, `omp_destroy_nest_lock()`
Lock example

- conflicts are rare, but to play it safe, we must assure mutual exclusion for updates to histogram elements.

```c
#pragma omp parallel for
for(i=0;i<NBUCKETS; i++){
    omp_init_lock(&hist_locks[i]); hist[i] = 0;
}
#pragma omp parallel for
for(i=0;i<NVALS;i++){
    ival = (int) sample(arr[i]);
    omp_set_lock(&hist_locks[ival]);
    hist[ival]++;
    omp_unset_lock(&hist_locks[ival]);
}
for(i=0;i<NBUCKETS; i++)
    omp_destroy_lock(&hist_locks[i]);
```

- One lock per element of hist
- Enforce mutual exclusion on update to hist array
- Free up storage when done
OpenMP runtime library routines

• Runtime environment routines:
  – Modify/Check the number of threads
    • `omp_set_num_threads()`, `omp_get_num_threads()`, `omp_get_thread_num()`, `omp_get_max_threads()`
  – Are we in an active parallel region?
    • `omp_in_parallel()`
  – Do you want the system to dynamically vary the number of threads from one parallel construct to another?
    • `omp_set_dynamic`, `omp_get_dynamic()`
  – How many processors in the system?
    • `omp_num_procs()`
OpenMP environment variables

- Set the default number of threads to use.
  - `MP_NUM_THREADS int_literal`
- OpenMP added an environment variable to control the size of child threads’ stack
  - `OMP_STACKSIZE`
- Also added an environment variable to hint to runtime how to treat idle threads
  - `OMP_WAIT_POLICY`
    - `ACTIVE` keep threads alive at barriers/locks
    - `PASSIVE` try to release processor at barriers/locks
- Process binding is enabled if this variable is true … i.e. if true the runtime will not move threads around between processors.
  - `OMP_PROC_BIND true | false`
OpenMP Tasks

- Tasks are independent units of work.
- Tasks are composed of:
  - code to execute
  - data environment
  - internal control variables (ICV)
- Threads perform the work of each task.
- The runtime system decides when tasks are executed
  - Tasks may be deferred
  - Tasks may be executed immediately
OpenMP Tasks (2)

- Task construct – `task` directive plus structured block
- Task – the package of code and instructions for allocating data created when a thread encounters a task construct
- Task region – the dynamic sequence of instructions produced by the execution of a task by a thread
Tasks are guaranteed to be complete at thread barriers:
\texttt{\#pragma omp barrier}

\texttt{\#pragma omp taskwait}

\texttt{\#pragma omp parallel}
\begin{verbatim}
{    \#pragma omp task
        foo();
    \#pragma omp barrier
    \#pragma omp single
    {        \#pragma omp task
                bar();
    }
}
\end{verbatim}
Data scoping with tasks

• A task’s private variables are undefined outside the task

```c
int fib ( int n )
{
    int x,y;
    if ( n < 2 ) return n;
#pragma omp task
    x = fib(n-1);
#pragma omp task
    y = fib(n-2);
#pragma omp taskwait
    return x+y; //!!!!!
}
```

```c
int fib ( int n )
{
    int x,y;
    if ( n < 2 ) return n;
#pragma omp task shared(x)
    x = fib(n-1);
#pragma omp task shared(y)
    y = fib(n-2);
#pragma omp taskwait
    return x+y;
}
```
OpenACC Step 4: Optimize

- Identify Parallelism
- Express Parallelism
- Express Data Locality
- Optimize

© Jeff Larkin, NVIDIA, S4167_JeffLarkin_Introduction_Compiler_Directives.pdf
Present clause

• It’s sometimes necessary for a data region to be in a different scope than the compute region.
• When this occurs, the present clause can be used to tell the compiler data is already on the device.
• Since the declaration of A is now in a higher scope, it’s necessary to shape A in the present clause.
• High-level data regions and the present clause are often critical to good performance.

```c
function main(int argc, char **argv)
{
    #pragma acc data copy(A)
    {
        laplace2D(A,n,m);
    }
}
```

```c
function laplace2D(double[N][M] A,n,m)
{
    #pragma acc data present(A[n][m])
    create(Anew)
    while ( err > tol && iter < iter_max ) {
        err=0.0;
        ...
    }
}
```
Clauses for the loop directive

- The `loop` directive provides the compiler with additional information for the next loop.

Notable Clauses:
- `private` & `reduction`
- `gang/worker/vector/seq`
- `collapse`
- `tile`
Private & reduction clause

- The private and reduction clauses are not optimization clauses, they may be required for correctness.

- **private** – A copy of the variable is made for each loop iteration (i.e. each thread)

- **reduction** – A reduction is performed on the listed variables.
  - Supports +, *, max, min, and various logical operations
Manual work layout

- OpenACC supports 3 levels of parallelism (gang, worker, vector).

**Parallel Loop**

```
#pragma acc parallel loop
num_gangs(X) num_workers(Y)
vector_length(Z)

#pragma acc loop seq
```

**Kernels**

```
#pragma acc kernels
#pragma acc loop gang(X)
#pragma acc loop worker(Y)
#pragma acc loop vector(Z)

#pragma acc loop seq
```
OpenACC supports three levels of parallelism

- **Vector** threads work in lockstep (SIMD/SIMT parallelism)
- **Workers** have 1 or more vectors.
- **Gangs** have 1 or more workers and share resources (such as cache, the streaming multiprocessor, etc.)
- Multiple gangs work independently of each other
- On NVIDIA GPUs, vector lengths should be a multiple of 32 and the worker dimension can usually be ignored.
The **collapse** clause instructs the compiler to convert the next N loops into one flattened loop.

- This is especially useful when some loops lack enough iterations to make effective use of the GPU.

```c
#pragma acc parallel loop
collapse(2)
for(int i=0; i<N; i++)
{
    for(int j=0; j<4; j++)
    {
        ...
    }
}
```

```c
#pragma acc parallel for
for(int ij=0; ij<4*N; ij++)
{
    ...
}
```
Tile clause

- The **tile** directive instructs the compiler to block the following loops to better exploit locality and data reuse.
  - The compiler will generate additional tile loops according to the tile dimensions provided.

```c
#pragma acc parallel loop
for(int i=0; i < n; i++)
  #pragma acc loop tile(8,8)
  for(int j=0; j < n; j++)
    for(int k=0; k < n; k++)
      c[i][j] = c[i][j] + 
      a[i][j] * b[k][j];
```
Summary of programming steps

1. **Identify Parallelism**
   - What important parts of the code have available parallelism?

2. **Express Parallelism**
   - Express as much parallelism as possible and ensure you still get correct results.
   - Because the compiler must be cautious about data movement, the code will generally slow down.

3. **Express Locality**
   - The programmer will always know better than the compiler what data movement is unnecessary.

4. **Optimize**
   - Don’t try to optimize a kernel that runs in a few us or ms until you’ve eliminated the excess data motion that is taking many seconds.
Async clause

• An async clause may be added to most OpenACC directives, causing the affected section of code to run asynchronously with the host.

• async(handle) – Add this directive to the asynchronous queue represented by the integer handle.
  – Asynchronous activities on the same queue execute in order
  – The user must wait before accessing variables associated with the asynchronous region.
  #pragma acc parallel loop async(1)
Wait directive

The wait directive instructs the compiler to wait for asynchronous work to complete.

```
#pragma acc wait(1)
```

- **wait** – Wait on all asynchronous work to complete
- **wait(handle)** – Wait strictly for the asynchronous queue represented by handle to complete.
- **wait(handle1) async(handle2)** – The asynchronous queue represented by handle2 must wait for all work in handle1 to complete before proceeding, but control is returned to the CPU.
Software pipelining with OpenACC

```c
#pragma acc data
for(int p = 0; p < nplanes; p++)
{
    #pragma acc update device(plane[p])
    #pragma acc parallel loop
    for (int i = 0; i < nwork; i++)
    {
        // Do work on plane[p]
    }
    #pragma acc update self(plane[p])
}
```

For this example, assume that each “plane” is completely independent and must be copied to/from the device.

As it is currently written, plane[p+1] will not begin copying to the GPU until plane[p] is copied from the GPU.
Software pipelining with OpenACC (2)

P and P+1 Serialize

P and P+1 Overlap Data Movement

NOTE: In real applications, your boxes will not be so evenly sized.
Software pipelining with OpenACC (2)

```c
#pragma acc data
for(int p = 0; p < nplanes; p++)
{
    #pragma acc update device(plane[p]) async(p)
    #pragma acc parallel loop async(p)
    for (int i = 0; i < nwork; i++)
    {
        // Do work on plane[p]
    }
    #pragma acc update self(plane[p]) async(p)
}
#pragma acc wait
```

Enqueue each plane in a queue to execute in order
Wait on all queues.

© Jeff Larkin, NVIDIA, Advanced Accelerated Computing Using Directives
Pipelined boundary exchange

Some algorithms, such as this boundary exchange, have a natural pipeline.

Each boundary zone is exchanged with one neighbor, so a separate queue may be used for each zone.
OpenACC and 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 );`
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
// use them e.g. for CUDA kernels
}
// x and y are host pointers
}
```

X and Y are device pointers here
CUDA Library calls from OpenACC

- OpenACC can interface with existing GPU-optimized libraries (from C/C++ or Fortran).

- This includes...
  - CUBLAS
  - Libscι_αcc
  - 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();
```

© Jeff Larkin, NVIDIA, Advanced Accelerated Computing Using Directives
Deviceptr clause

- 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
cudaMalloc((void*) &x, (size_t)n*sizeof(float));
cudaMalloc((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.
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

© Jeff Larkin, NVIDIA, Advanced Accelerated Computing Using Directives
QUESTIONS?