GPU COMPUTING
LECTURE 10 - PRODUCTIVITY

Kazem Shekofteh
kazem.shekofteh@ziti.uni-heidelberg.de
Institute of Computer Engineering
Ruprecht-Karls University of Heidelberg
Inspired from lectures by Holger Fröning

With material from Sandra Wienke, RWTH Aachen (OpenACC)

OPENACC - DIRECTIVE-BASED ACCELERATOR PROGRAMMING

OVERVIEW

Up to now: CUDA (& OpenCL)

- No major differences, both regarding usage and performance
- Quite unproductive development process
- As for any imperative programming language

Alternative: directive-based programming

- Compiler responsible for low-level implementation
- Kernel execution, data movements, optimizations, …
- Declarative programming language (like OpenMP)

OpenACC: directive-based programming model to off-load compute-intensive loops to accelerators

OVERVIEW

Goal: simplify parallel programming of heterogeneous CPU/GPU systems

Open industry standard

- Cross-platform, C/C++/Fortran
- Several vendors
- Currently: Cray, CAPS, OpenARC, NVIDIA, PGI
- First specification in 2011, first compilers in 2012

Another objective: mid-term integration into OpenMP resp. extending OpenMP by lessons learned with OpenACC

EXECUTION MODEL

Execution model: host-directed execution with an attached accelerator device

- Offloading compute-intensive regions (parallel region or kernel region)
- Three parallelism levels

Coarse-grain parallelism: fully parallel execution across execution units -> **gang parallelism**

- Limited support for synchronization
- CUDA: multiple thread blocks -> grid level

Fine-grain parallelism: multiple threads on a single execution unit -> **worker parallelism**

- Latency hiding techniques
- CUDA: warps at block level

SIMD/vector operations: multiple operations per thread -> **vector parallelism**

- CUDA: threads at block level

Programmer has to identify appropriate parallelism type

- Fully-parallel loop (no dependencies) -> gang
- Vectorizable loop but with dependencies -> fine-grain parallelism, or sequential implementation

MEMORY MODEL

Accelerator may be completely separate from host memory -> explicit data movement using DMA

- CUDA/OpenCL: user is responsible and resulting code can be complex

Memory model for OpenACC

- Compiler is responsible for data movements (with the help of directives)

Issues: PCI: bandwidth disparity (computational intensity), limited device memory, dereferencing host/device pointers

- Weak consistency model
- Support for software-controlled caches (scratchpad): SHARED MEM
- Managed by compiler instead of programmer (as for CUDA/OpenCL)

DIRECTIVES

Simplified execution and memory model

-> Many tasks can be done by compiler/runtime

-> User-directed programming

Iterative development process

Compiler provides helpful feedback about implementation

If accelerator kernels could be generated

Which loop schedule is used

Data movement

C

```
#pragma acc directive-name [clauses]
```

Fortran

```
!$acc directive-name [clauses]
```

EXAMPLE: SAXPY – SERIAL (HOST)

```
int main() {
  int n = 256*1024*1024; float a = 2.0f; float b = 3.0f;
  float* restrict x; float* restrict y;
  // Allocate & initialize x, y
  <snip>



  for (int i = 0; i < n; ++i) {
    y[i] = a*x[i] + y[i];
  }

  for (int i = 0; i < n; ++i) {
    y[i] = b*x[i] + y[i];
  }


  //free and cleanup
  <snip>
}
```

EXAMPLE: SAXPY – OPENACC #1

```
int main() {
  int n = 256*1024*1024; float a = 2.0f;
  float* restrict x; float* restrict y;
  // Allocate & initialize x, y
  <snip>

#pragma acc kernels
{                                          //26
  for (int i = 0; i < n; ++i) {            //27
    y[i] = a*x[i] + y[i];
  }

  for (int i = 0; i < n; ++i) {            //31
    y[i] = b*x[i] + y[i];
  }
}

  //free and cleanup
  <snip>
}
```

Without async clause: implicit barrier at the end

GCC COMPILER FEEDBACK

```
$ gcc -Wall -O2 openacc-saxpy_01.c -o openacc-saxpy_01gcc
openacc-saxpy_01.c:25: warning: ignoring #pragma acc kernels [-Wunknown-pragmas]
```

=> exactly same as without pragmas

PGI COMPILER FEEDBACK

```
$ pgc++ -O2  -acc -ta=tesla,time -Minfo=accel -g openacc-saxpy_01.c -o openacc-saxpy_01pgc
main:
     26, Generating implicit copyin(x[:268435456]) [if not already present]
         Generating implicit copy(y[:268435456]) [if not already present]
     27, Loop is parallelizable
         Generating Tesla code
         27, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     31, Loop is parallelizable
         Generating Tesla code
         31, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

EXAMPLE: SAXPY – OPENACC #1

```
int main() {
  int n = 256*1024*1024; float a = 2.0f; float b = 3.0f;
  float* restrict x; float* restrict y;
  // Allocate & initialize x, y
  <snip>

#pragma acc kernels
{                                          //26
  for (int i = 0; i < n; ++i) {            //27
    y[i] = a*x[i] + y[i];
  }

  for (int i = 0; i < n; ++i) {            //31
    y[i] = b*x[i] + y[i];
  }
}                                          //34

  //free and cleanup
  <snip>
}
```

```
$ ./openacc-saxpy_01pgc
<snip>
    26: compute region reached 1 time
        27: kernel launched 1 time
            grid: [65535]  block: [128]
            elapsed time(us): total=6,005 <snip>
        31: kernel launched 1 time
            grid: [65535]  block: [128]
            elapsed time(us): total=5,987 <snip>
    26: data region reached 2 times
        26: data copyin transfers: 128
             device time(us): total=171,564 <snip>
        34: data copyout transfers: 65
             device time(us): total=82,314 <snip>
```

EXAMPLE: SAXPY – OPENACC #2

```
<snip>                                    //21
#pragma acc parallel loop
  for (int i = 0; i < n; ++i) {           //26
    y[i] = a*x[i] + y[i];
  }

#pragma acc parallel loop                 //30
  for (int i = 0; i < n; ++i) {           //31
    y[i] = b*x[i] + y[i];
  }
```

Parallel construct: each gang executes the code in gang-redundant mode

-> code within the parallel region, but outside of a loop with a loop directive and gang-level work-sharing, will be executed redundantly by all gangs

PGI COMPILER FEEDBACK

```
    21, Generating Tesla code
        26, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    21, Generating implicit copyin(x[:268435456]) [if not already present]
        Generating implicit copy(y[:268435456]) [if not already present]
    28, Generating Tesla code
        31, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    28, Generating implicit copy(y[:268435456]) [if not already present]
        Generating implicit copyin(x[:268435456]) [if not already present]
```

EXAMPLE: SAXPY - OPENACC #3

```
#pragma acc data copyin(x[0:n]) copy(y[0:n])
{                                              //25
#pragma acc parallel loop present(x,y)
  for (int i = 0; i < n; ++i) {                //27
    y[i] = a*x[i] + y[i];
  }
#pragma acc parallel loop present(x,y)         //31
  for (int i = 0; i < n; ++i) {                //32
    y[i] = b*x[i] + y[i];
  }
}
```

PGI COMPILER FEEDBACK

```
     25, Generating copy(y[:n]) [if not already present]
         Generating copyin(x[:n]) [if not already present]
         Generating present(y[:],x[:])
         Generating Tesla code
         27, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     29, Generating present(y[:],x[:])
         Generating Tesla code
         32, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

EXAMPLE: SAXPY - OPENACC #4

```
#pragma acc data copyin(x[0:n]) copy(y[0:n])
{
#pragma acc parallel loop vector_length(256)
  for (int i = 0; i < n; ++i) {
    y[i] = a*x[i] + y[i];
  }
#pragma acc parallel loop vector_length(256)
  for (int i = 0; i < n; ++i) {
    y[i] = b*x[i] + y[i];
  }
}
```

PGI COMPILER FEEDBACK

```
    25, Generating copyin(x[:n]) [if not already present]
        Generating copy(y[:n]) [if not already present]
        Generating Tesla code
        27, #pragma acc loop gang, vector(256) /* blockIdx.x threadIdx.x */
    29, Generating Tesla code
        32, #pragma acc loop gang, vector(256) /* blockIdx.x threadIdx.x */
```

PERFORMANCE RESULTS

	CPU (00)	ACC01	ACC02	ACC03	ACC04	ACC01UM
kernel1 [ms]		6,0	6,0	6,0	5,8	372,5
kernel2 [ms]		6,0	6,0	6,0	5,8	5,9
sum copies [ms]		253,9	507,4	253,7	253,8	0,0
total [ms]	**286,5**	**265,9**	**519,4**	**265,7**	**265,4**	**378,4**
GB/s (256M elem)	3,7	4,0	2,1	4,0	4,0	2,8

How to use UM: `-ta=tesla:managed,time`

There's more: `{pinned, managed, autocompare}`

DIRECTIVES: OFFLOAD REGION

Offload Region <=> CUDA Kernel function

Parallel

- User responsible to find parallelism (loops)
- More explicit
- `acc loop` directive mandatory
- No automatic synchronization between loops

Kernels

- Compiler responsible to find parallelism (loops)
- More implicit, more compiler freedom
- `acc loop` directive only for performance tuning
- Automatic synchronization between loops

```
#pragma acc parallel [clauses]
{
  <seq. code>
  for ( int i = 0; i < n; ++i ) {
  }

  for ( int i = 0; i < n; ++i ) {
  }
}
```

```
#pragma acc kernels [clauses]
{
 <seq. code>
  for ( int i = 0; i < n; ++i ) {
  }

  for ( int i = 0; i < n; ++i ) {
  }
}
```

THE PARALLEL DIRECTIVE IN DETAIL

1. Distributes outer loop to n threads
   - Each thread executes inner loop sequentially
   - 256 threads per block
   - n/256 (rounded up) blocks
2. Similar to above, but fixed number of blocks (16)
   - 256 threads per block
   - If 16*256 < n then each thread gets multiple loop iterations
3. Now parallelizing both loops
   - Distributing n outer loops to n blocks
   - Distributing inner loop to threads within these blocks
   - 256 threads per block

```
#pragma acc parallel vector_length(256)
#pragma acc loop gang vector
  for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < m; ++j ) {
      // do something
    }
  }
```

```
#pragma acc parallel vector_length(256)
                    num_gangs(16)
#pragma acc loop gang vector
  for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < m; ++j ) {
      // do something
    }
  }
```

```
#pragma acc parallel vector_length(256)
#pragma acc loop gang
  for ( int i = 0; i < n; ++i ) {
#pragma acc loop vector
    for ( int j = 0; j < m; ++j ) {
      // do something
    }
  }
```

DIRECTIVES: LOOPS

A loop is shared among different sets of threads, or is executed sequentially (schedule parameter)

Loop schedules (clauses)

Specify the mapping of loop-level parallelism to OpenACC abstractions

```
#pragma acc loop [clauses]
{
    ...
}
```

Clause	Description
gang	Distribute work among thread **blocks**
worker	Distribute work among thread **warps** of a block
vector	Distribute work among **threads**
seq	Execute loops sequentially

DIRECTIVES: DATA

Data region directives are used to decouple data movement from offload regions

Data clauses describe when data movement is actually necessary

```
#pragma acc data [clauses]
{
    ...
}
```

Clause	Description
copy	H2D at region start, D2H at region end
copyin	Only H2D at region start
copyout	Only D2H at region end
create	Allocate data on the device, no data transfer
present	Data is already allocated on the device

DIRECTIVES: MISCELLANEOUS

Reduction clause: code generated by compiler

- For parallel and loop constructs; doesn't replace code
- Operators: +, *, min, max, …

```
#pragma parallel/kernel loop reduction(op:list)
```

Update directive

- Updates the content of arrays that exist on host and device
- Either update the host side or the device side

```
#pragma acc data copy(x[0:n])...
{
  for( timestep=0;... ) {
    ...compute on device...
    #pragma update host (x[0:n])
    MPI_SENDRECV( x, ... )
    #pragma update device (x[0:n])
    ...use on device...
  }
}
```

DIRECTIVES: MISCELLANEOUS

What‘s missing up to now?

Cache construct

- Prioritizes data for placement in the highest cache hierarchy level
- GPUs: cache refers to shared memory (scratchpad)
- Caches are managed by the compiler with hints by the programmer

Possibly not fully implemented for some compilers

Possibly not working as intended if cache is oversubscribed

```
#pragma acc cache ( list )

Example:
for ( start = 0; start < max; start += length ) {
  #pragma acc cache ( array [start:length-1] )
  ...
}
```

OPTIMIZATIONS

What if `ptrA` points to the same location that `val points`?

Problem: compiler cannot determine if pointers used in different loop iterations are independent

-> could be pointer aliasing

-> prevents parallelization

If no dependencies are actually present:

- Use restrict keyword for variable declaration: `float* restrict ptr;`
- Use complier flags (carefully): `-anti-alias`
- Use OpenACC loop clause: `independent`

```
void updatePtrs (size_t *ptrA,
                 size_t *ptrB, size_t *val)
  {
  *ptrA += *val;
  *ptrB += *val;
  }
```

```
#pragma acc kernels loop independent
for( i = 0; i < n; ++i )
  a[i] = b[i] + c[i];
```

Asynchronous Execution: use `async` clause for update and compute constructs (update, kernel, parallel)

- In-order if async integer parameter is the same
- Out-of-order/overlap allowed for different integer parameters

LAPLACE STENCIL CODE

by Mark Harris[1]

Sandy Bridge CPU vs. Pascal GPU

```
#pragma acc data copy(A, Anew)
while ( err > tol && iter < iter_max ) {
  err = 0.f;

  #pragma omp parallel for shared(m, n, Anew, A)
  #pragma acc kernels
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      Anew[j][i]=0.25f*(A[j][i+1] + A[j][i-1]
                        + A[j-1][i] + A[j+1][i]);
      err=fmaxf(error,fabsf(Anew[j][i]-A[j][i]));
      }
    }

  #pragma omp parallel for shared(m, n, Anew, A)
  #pragma acc kernels
  for( int j = 0; j < n-1; j++)
    for( int i = 0; i < m-1; i++ )
      A[j][i] = Anew[j][i];
  iter++;
}
```

```
OMP
#threads     time
1          69.467537s
2          36.694647s
4          35.993741s
8          38.307806s

ACC
version  time total        copy     comment
1           150.89s      97.04%    data copy in between each kernel iteration
2             1.39s      12.73%    avoid such data copies: #pragma acc data ...
3             1.06s       8.14%    kernel launch configuration optimizations
```

[1] https://devblogs.nvidia.com/openacc-example-part-1/

WRAPPING UP

OpenACC

SUMMARY

OpenACC: directive-based constructs and tuning capabilities for parallelization on GPUs

- Productive development process, but comprehensive knowledge about the GPU architecture and execution model still required
- Compiler feedback can be cryptic
- Performance likely won't be top-notch
- Don't over-constrain: otherwise performance portability might be limited

Libraries allow to utilize hand-optimized code implementations

- Such as cuBLAS (linear algebra), thrust (parallel algorithms and data structures, similar to STL)
- Minimal effort, but deep understanding of library operators required
- Compilers can't reason about (non-header-only) library calls
- Various other libraries: cuDNN, cuFFT, cuRAND, cuSOLVER, cuSPARSE, nvGRAPH, gunrock, cuGraph, …

ACCELERATION LIBRARIES

CUDA THRUST

Thrust is a C++ template library for CUDA based on the Standard Template Library (STL)

- Performance-portable abstraction layer
- High-level parallel algorithm library
- Data parallel primitives such as scan, sort, and reduce

Productivity

- Containers: `host_vector, device_vector`
- Memory management: data transfers
- Algorithm selection: implicit location

Interoperability

- CUDA, OpenMP, Intel TBB
- E.g.: `thrust::omp::vector`

CUDA THRUST - SORTING EXAMPLE

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <cstdlib>

int main ()
  {
  // generate 32M random numbers on the host
  thrust::host_vector < int > h_vec ( 32 << 20 );
  thrust::generate ( h_vec.begin (), h_vec.end(), rand );

  // transfer data to the device
  thrust::device_vector < int > d_vec = h_vec ;

  // sort data on the device
  thrust::sort ( d_vec.begin(), d_vec.end() );

  // transfer data back to host
  thrust::copy ( d_vec.begin(), d_vec.end(), h_vec.begin() );
  return 0;
}
```

CUDA THRUST - INTEROPERABILITY

```
thrust::omp::vector <float> my_omp_vec ( 100 );
thrust::cuda::vector <float> my_cuda_vec ( 100 );

...

// reduce in parallel on the CPU
thrust::reduce ( my_omp_vec.begin(), my_omp_vec.end() );

//sort in parallel on the GPU
thrust::sort ( my_cuda_vec.begin(), my_cuda_vec.end() );
```

CUBLAS

The CUBLAS library is an implementation of BLAS (Basic Linear Algebra Subprograms) using CUDA

Levels 1 – 3

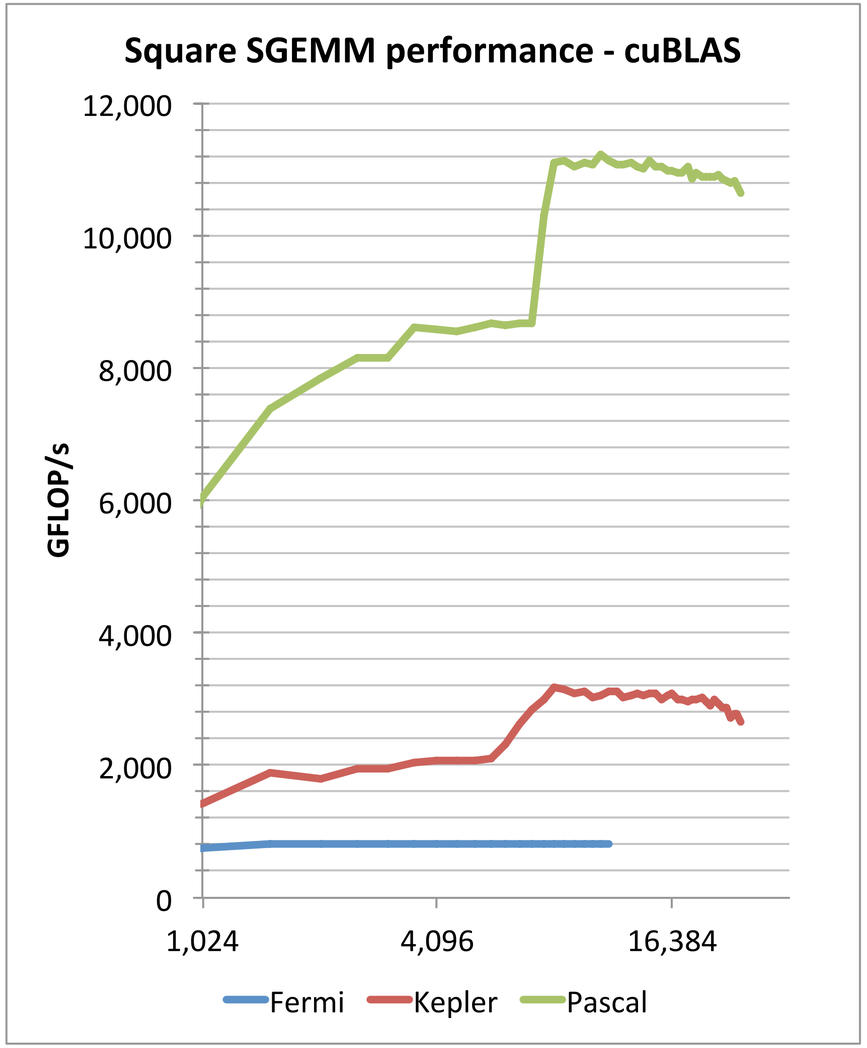
Vector-vector, vector-matrix, matrix-matrix

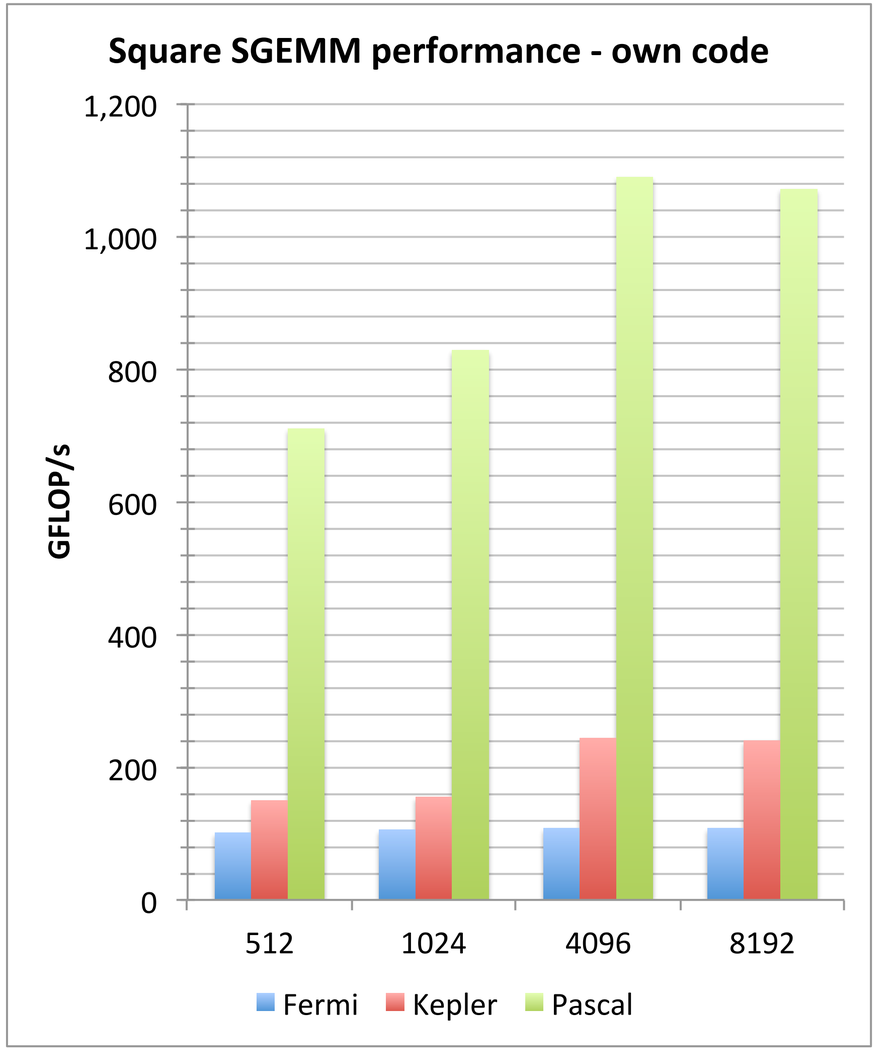
Example: single precision matrix multiply

$$C = \alpha \cdot op(A) \cdot op(B) + \beta \cdot C$$

```
cublasStatus_t cublasSgemm ( cublasHandle_t handle,
                             cublasOperation_t transa, cublasOperation_t transb,
                             int m, int n, int k,
                             const float           *alpha,
                             const float           *A, int lda,
                             const float           *B, int ldb,
                             const float           *beta,
                             float                 *C, int ldc )
```

CUBLAS


Square SGEMM performance - cuBLAS
GFLOP/s
12,000
10,000
8,000
6,000
4,000
2,000
0
1,024
4,096
16,384
Fermi
Kepler
Pascal


Square SGEMM performance - own code
GFLOP/s
1,200
1,000
800
600
400
200
0
512
1024
4096
8192
Fermi
Kepler
Pascal