OpenACC

Ned Nedialkov

McMaster University

21 March 2023

Outline

Why accelerators

- If a program execution cannot fit on a single machine and/or many processors are needed: go distributed
  Message-Passing Interface (MPI)
- If shared memory would do: OpenMP or Pthreads
- Cheaper alternative: accelerators
  - GPUs (NVIDIA, AMD ...)
  - Intel Xeon Phi
- GPUs are not easy to program
  - CUDA supports NVIDIA only
  - OpenCL is portable, harder than CUDA
  - OpenACC
    - Portable, do not need to know much about the hardware
    - Much easier than CUDA and OpenCL

OpenACC overview

- Set of compiler directives, library routines, and environment variables
- Fortran, C, C++
- Initially developed by PGI, Cray, NVIDIA, CAPS
  OpenACC 1.0 in 2011
  Latest standard 3.3
- Done through pragmas
- We can annotate a serial program with OpenACC directives
  Non-OpenACC compilers can simply ignore the pragmas

References

- OpenACC web site `http://www.openacc.org/`
- Kirk & Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*
- PGI Accelerator Compilers. OpenACC Getting Started Guide `https://www.pgroup.com/doc/openacc_gs.pdf`
- PGI compiler and tools `https://www.pgroup.com/resources/articles.htm`
- OpenACC quick reference `http://www.nvidia.com/docs/IO/116711/OpenACC-API.pdf`
- 11 Tips for Maximizing Performance with OpenACC Directives in Fortran `https://www.pgroup.com/resources/openacc_tips_fortran.htm`

OpenACC example: matrix-matrix multiplication

```
1  #ifdef _OPENACC
2  #include <openacc.h>
3  #endif
4
5  /* A is m x n, B is n x p, C = A*B is m x p */
6  void matmul_acc(float *restrict C, float *restrict A, float *restrict B, int m,
                   int n, int p) {
7      int i, j, k;
8  #pragma acc kernels copyin(A[0:m * n], B[0:n * p]) copyout(C[0:m * p])
9      {
10         for (i = 0; i < m; i++)
11             for (j = 0; j < p; j++) {
12                 float sum = 0;
13                 for (k = 0; k < n; k++)
14                     sum += A[i * n + k] * B[k * p + j];
15                 C[i * p + j] = sum;
16             }
17     }
18 }
```

Execution model

An OpenACC program starts as a single thread on the host

- parallel or kernels construct identify parallel or kernels region
- when the program encounters a parallel construct, gangs of workers are created to execute it on the accelerator
- one worker, the gang leader, starts executing the parallel region
- work is distributed when a work-sharing loop is reached

Three levels of parallelism: gang, worker, vector

- a group of gangs execute a kernel
- a group of workers can execute a work-sharing loop from a gang
- a thread can execute vector operations

Memory model

- Main memory and device memory are separate
- Typically
  - transfer memory from host to device
  - execute on device
  - transfer result to host

Compiling

- The ▸ Nvidia compilers support OpenACC
- ▸ GCC 12, 11, 10: support OpenACC 2.6
- To compile with `nvc`,

```
nvc -fast -acc -Minfo -gpu=cc60  \
                      -c -o matmul_acc.o matmul_acc.c
```

  - `-fast` create generally an optimal set of flags
  - `-acc` generate accelerator code
  - `-Minfo` output compiler info
  - `-gpu=...` GPU type

    To find out the GPU type, run `pgaccelinfo`. Here it gives
    `Default Target:  cc60`

Compiling with `-Minfo` outputs

```
matmul_acc:
      9, Generating copyin(A[:n*m]) [if not already present]
         Generating copyout(C[:m*p]) [if not already present]
         Generating copyin(B[:n*p]) [if not already present]
     10, Loop carried dependence of C-> prevents parallelization
         Loop carried backward dependence of C-> prevents vectorization
         Loop not fused: no successor loop
     11, Loop is parallelizable
         Generating NVIDIA GPU code
         10, #pragma acc loop seq
         11, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         13, #pragma acc loop seq
     13, Loop is parallelizable
         Generated vector simd code for the loop containing reductions
     14, FMA (fused multiply-add) instruction(s) generated
```

OpenMP example

```
#ifdef _OPENMP
#include <omp.h>
#endif

void matmul_mp(float *restrict C, float * restrict A, float * restrict B, int m,
               int n, int p) {
    int i, j, k;

#pragma omp parallel shared(A, B, C) private(i, j, k)
    {
#pragma omp for schedule(static)
        for (i = 0; i < m; i++)
            for (j = 0; j < p; j++) {
                float sum = 0;
                for (k = 0; k < n; k++)
                    sum += A[i * n + k] * B[k * p + j];
                C[i * p + j] = sum;
            }
    }
}
```

Speedups

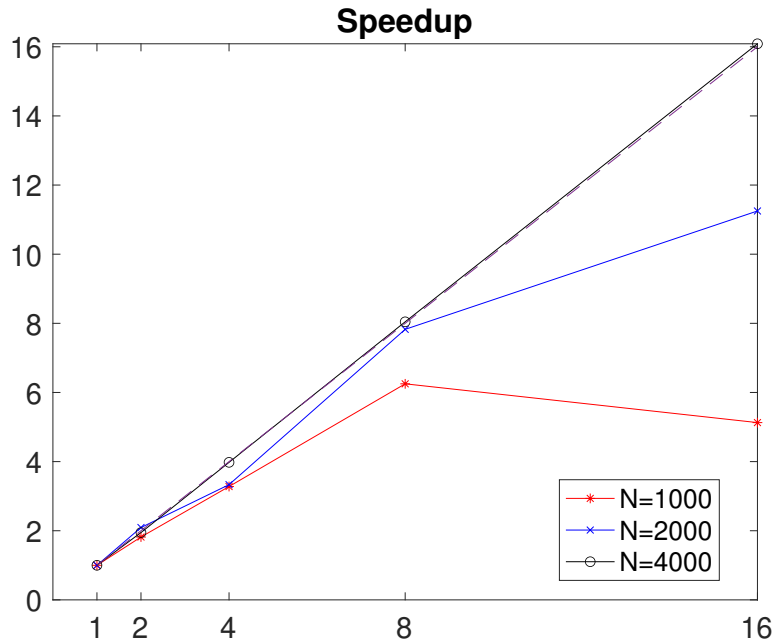
Speedup results on Sharcnet's `graham` 852

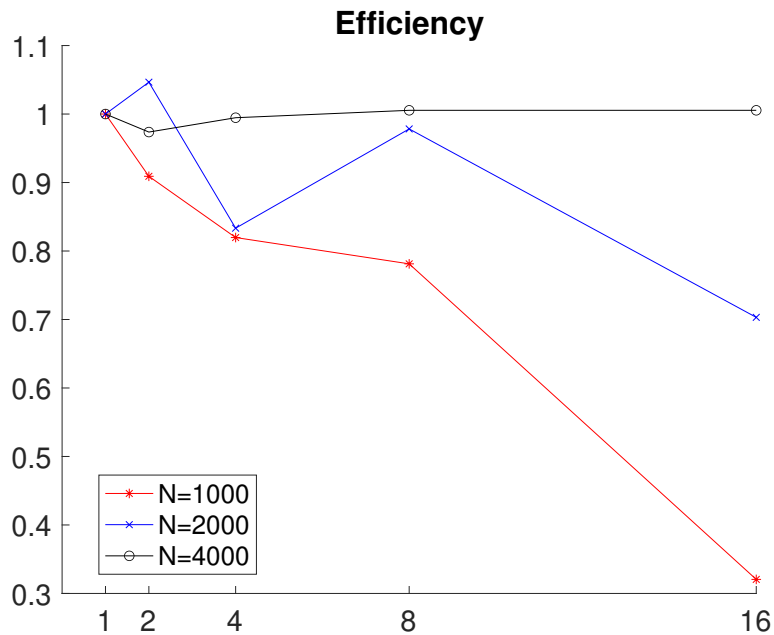
```
32 cores
2 sockets x 16 cores per socket
Intel E5-2683 v4 (Broadwell) @ 2.1 GHz
2 × NVIDIA Pascal P100 GPUs (12GB HBM2)
Memory: 128.0 GB
```

Speedups

	# threads	secs/speedup compared to 1 core		
		N		
	p	1000	2000	4000
OpenMP	1	2.0e+00/1.0	1.8e+01/1.0	3.7e+02/1.0
	2	1.1e+00/1.8	8.6e+00/2.1	1.9e+02/1.9
	4	6.1e-01/3.3	5.4e+00/3.3	9.3e+01/4.0
	8	3.2e-01/6.2	2.3e+00/7.8	4.6e+01/8.0
	16	3.9e-01/5.1	1.6e+00/11.2	2.3e+01/16.1
GPU		2.1e-01/9.5	3.9e-01/46.2	1.2e+00/308.3

Speedup and efficiency of OpenMP code





PGI_ACC_TIME

To output profiling information, set in Bash

```
export PGI_ACC_TIME=1
```

Executing `./matmul_acc 1000` gives

```
Accelerator Kernel Timing data
/home/ned/gpu/matmul_acc.c
  matmul_acc  NVIDIA  devicenum=0
    time(us): 1,050
    11: compute region reached 1 time
        14: kernel launched 1 time
            grid: [8]  block: [128]
            elapsed time(us): total=50,370 max=50,370 min=50,370
                              avg=50,370
    11: data region reached 2 times
        11: data copyin transfers: 2
             device time(us): total=725 max=376 min=349 avg=362
        22: data copyout transfers: 1
             device time(us): total=325 max=325 min=325 avg=325
```

- block: a a group of threads that are scheduled to execute together
- grid: collection of blocks that are scheduled to execute
- Here 8 blocks $\times$ 128 threads each $=$ 1024 threads

CUDA

CUDA: Compute Unified Device Architecture

Kernel

- function running on the GPU
- executed by a (1D or 2D) grid of thread blocks
- thread blocks can be 1D, 2D or 3D
  - execute independently of each other
  - threads within a single thread block can synchronize
- grid size and thread block size are defined when a kernel is launched

Programming

- NVIDIA GPUs are programmed as a sequence of kernels
- typically, a kernel completes execution before the next kernel begins
- threads are grouped into blocks, and blocks are grouped into a grid
- a kernel is executed as a grid of blocks of threads
- a thread has a unique local index in its block
- a block has a unique index in the grid

- number of gangs and number of workers in each gang remain constant in a parallel region
- `num_gangs` clause specifies number of gangs
- `num_workers` clause specifies number of workers within each gang
- `vector_length` clause specifies vector length for SIMD operations within each worker of the gang