OpenACC. Part I

Ned Nedialkov

McMaster University
Canada

October 2016

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, ATI ...)
  - 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
    - Still not trivial to accelerate code with OpenACC

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, 2.5 in 2015
- Done through pragmas
  A pragma is a directive to the compiler and contains information not specified in the language
- 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
- Tesla vs. Xeon Phi vs. Radeon. A Compiler Writer's Perspective http://www.pgroup.com/lit/articles/insider/v5n2a1.htm
- 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  void matmul_acc(float * restrict C, float * restrict A,
6                  float * restrict B, int m, int n, int p)
7  {
8    /* A is m x n, B is n x p, C = A*B is m x p */
9    int i,j,k;
10 #pragma acc kernels copyin(A[0:m*n], B[0:n*p])  copyout(C[0:m*p])
11   {
12     for (i=0; i<m; i++)
13       for (j=0; j<p; j++)
14         {
15           float sum = 0;
16           for (k=0; k<n; k++)
17             sum += A[i*n+k]*B[k*p+j];
18
19           C[i*p+j] = sum;
20         }
21   }
22 }
```

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 pgcc compiler appears to have the best support for OpenACC
- gcc has limited support
  https://gcc.gnu.org/wiki/OpenACC
- We have pgcc version 14.10, 2014, see the servers with GPUs at
  http://www.cas.mcmaster.ca/support/index.php/Servers
  14.10 has glitches, but OK
- To compile

```
pgcc -fast -acc -Minfo -ta=tesla,cc35  \
                       -c -o matmul_acc.o matmul_acc.c
```

  - `-fast` create generally an optimal set of flags
  - `-acc` generate accelerator code
  - `-Minfo` output useful info
  - `-ta=` architecture; important to get it right
    Run `pgaccelinfo` to get it

Compiling outputs

```
pgcc -fast -acc -Minfo -ta=tesla,cc35   -c -o matmul_acc.o matmul_acc.c
matmul_acc:
     10, Generating copyin(A[:n*m])
         Generating copyin(B[:n*p])
         Generating copyout(C[:m*p])
         Generating Tesla code
     12, Loop carried dependence of 'C->' prevents parallelization
         Loop carried backward dependence of 'C->' prevents vectorization
     13, Loop is parallelizable
         Accelerator kernel generated
         13, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     16, Loop is parallelizable
```

pgaccelinfo

If a GPU is set up properly, `pgaccelinfo` gives information like

```
CUDA Driver Version:           6050
NVRM version:                  NVIDIA UNIX x86_64 Kernel Module  340.29  Thu Jul

Device Number:                 0
Device Name:                   Tesla K40c
Device Revision Number:        3.5
Global Memory Size:            12079136768
Number of Multiprocessors:     15
Number of SP :           2880
Number of DP Cores:            960
Concurrent Copy and Execution: Yes
Total Constant Memory:         65536
Total Shared Memory per Block: 49152
Registers per Block:           65536
Warp Size:                     32
Maximum Threads per Block:     1024
Maximum Block Dimensions:      1024, 1024, 64
Maximum Grid Dimensions:       2147483647 x 65535 x 65535
Maximum Memory Pitch:          2147483647B
Texture Alignment:             512B
Clock Rate:                    745 MHz
```

```
Execution Timeout:              No
Integrated Device:              No
Can Map Host Memory:            Yes
Compute Mode:                   default
Concurrent Kernels:             Yes
ECC Enabled:                    Yes
Memory Clock Rate:              3004 MHz
Memory Bus Width:               384 bits
L2 Cache Size:                  1572864 bytes
Max Threads Per SMP:            2048
Async Engines:                  2
Unified Addressing:             Yes
Initialization time:            1961606 microseconds
Current free memory:            11976704000
Upload time (4MB):              1636 microseconds (1382 ms pinned)
Download time:                  2727 microseconds (1276 ms pinned)
Upload bandwidth:               2563 MB/sec (3034 MB/sec pinned)
Download bandwidth:             1538 MB/sec (3287 MB/sec pinned)
PGI Compiler Option:            -ta=tesla:cc35
```

OpenMP example

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

void matmul_mp(float * C, float * A, float * B,
               int m, int n, int p)
{
  /* A is m x n, B is n x p, C = A*B is m x 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;
        }
  }
}
```

Main program

```
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>

#ifdef _OPENACC
#include <openacc.h>
extern void  matmul_acc(float * restrict C, float * restrict A,
                        float * restrict B, int m, int n, int p);
#endif

#ifdef _OPENMP
#include <omp.h>
extern void  matmul_mp(float * C, float * A, float * B,
                       int m, int n, int p);
#endif
```

```
int main(int argc, char *argv[])
{
  int i,N;
  float *A, *B, *C;
  double time;

  if (argc==2)  sscanf(argv[1],"%d", &N);
  else {
    printf("Usage␣%s␣N␣\n", argv[0]);
    return 1;
  }

  A = (float*)malloc(N*N*sizeof(float));
  B = (float*)malloc(N*N*sizeof(float));
  C = (float*)malloc(N*N*sizeof(float));

  int num_threads=1;

  for (i=0;i<N*N;i++)     A[i] = i;
  for (i=0;i<N*N;i++)     B[i] = i;
```

```
#ifdef _OPENMP
#pragma omp parallel
  num_threads = omp_get_num_threads();
  time = omp_get_wtime();
  matmul_mp(C,A,B,N,N,N);
  time = omp_get_wtime()-time;
#endif

#ifdef _OPENACC
  struct timeval start, end;
  gettimeofday(&start, NULL);
  matmul_acc(C,A,B,N,N,N);
  gettimeofday(&end, NULL);
  time = end.tv_sec-start.tv_sec+(end.tv_usec - start.tv_usec)*1.e-6;
#endif

  printf("%d  %.1e\n", num_threads, time);

  free(C);
  free(B);
  free(A);
  return 0;
}
```

makefiles

```
# makefile for OpenMP
CC=pgcc
CFLAGS=-O2 -mp
LDFLAGS=-mp
matmul_mp: main.o matmul_mp.o
	$(CC) $(LDFLAGS) -o $@ $?

clean:
	rm *.o *~ matmul_mp

# makefile for OpenACC
CC=pgcc
CFLAGS=-fast -acc -Minfo -ta=tesla,cc30
LDFLAGS=-acc -ta=tesla,cc30

matmul_acc: main.o matmul_acc.o
	$(CC) $(LDFLAGS) -o $@ $?

clean:
	rm *.o *~ matmul_acc
```

Speedups

Speedup results on `tesla.mcmaster.ca`

- 2 Quad Core Xeon X5482 3.20GHz,
- `pgcc -O2 -mp`

and

- NVIDIA
- `pgcc -fast -acc`

	cores/peak	
	Single	Double
Tesla K40c	2880/4.29 TFLOPS	960/1.43 TFLOPS
Quadro K5000	1536/2.1 TFLOPS	512
Quadro K2000	384/732.7 GFLOPS	128

	# threads	secs/speedup compared to 1 core			
		$N = 800$	2000	4000	8000
OpenMP	1	0.84/	26/	170/	1400/
	2	0.73/1.2	13/2.0	110/1.5	870/1.6
	4	0.37/2.3	6.3/4.1	53/3.2	460/3.0
	8	0.20/4.2	3.4/7.6	29/5.9	240/5.8
	16	0.21/4.0	3.6/7.2	38/4.5	370/3.8
K40c		2.3/0.4	3.4/7.4	7.4/23.0	26/53.8
K2000		0.82/1.0	1.7/15.2	6.8/25.0	49/28.6
K5000		0.6/1.2	1.5/17.3	4.8/35.4	22/63.6

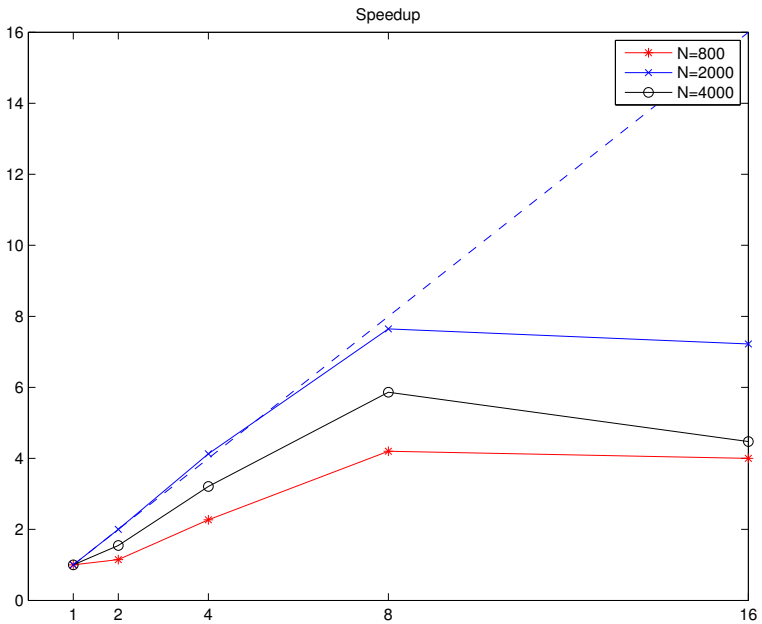
Speedups

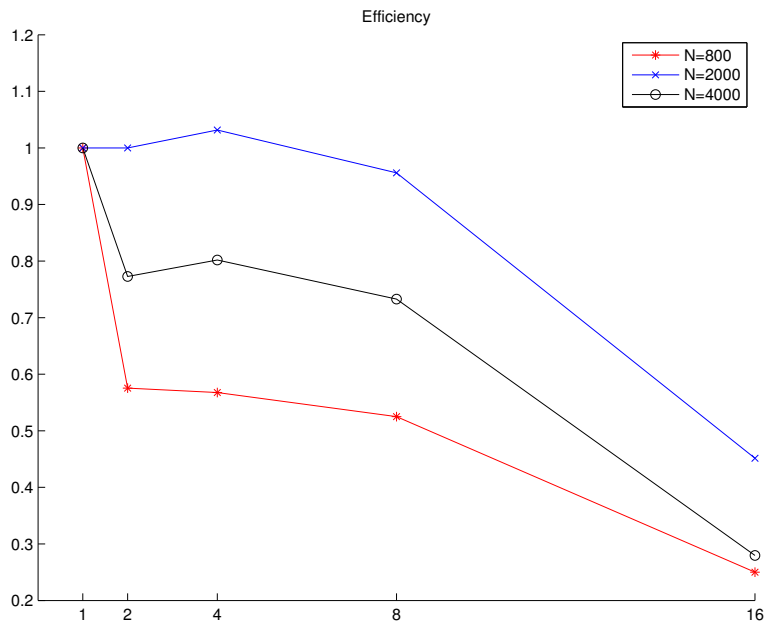
Speedup results on `advol3.mcmaster.ca`

- AMD 8 core Opteron 885, `gcc -O2 -fopenmp`
- K40c, K2000, K5000, `pgcc -fast -acc`

	# threads	secs/speedup compared to 1 core		
		$N = 800$	2000	4000
OpenMP	1	3.8/	42/	350/
	2	1.9/2.0	22/1.9	180/1.9
	4	1.1/3.5	13/3.2	100/3.5
	8	0.56/6.8	6.5/6.5	53/6.6
	16	0.36/10.6	5.1/8.2	40/8.8
K40c		2.3/0.7	3.4/9.4	7.4/31.2
K2000		0.82/4.6	1.7/24.5	6.8/51.5
K5000		0.6/6.3	1.5/28.0	4.8/72.9

Speedup and efficiency of OpenMP code





ACC_NOTIFY

To see if anything has executed on the GPU set before execution

```
csh: setenv ACC_NOTIFY 1
bash: export ACC_NOTIFY=1
```

You should see e.g.

```
[nedialk@gpu2 ~/GPU1/code] ./matmul_acc 2000
launch CUDA kernel  file=/nfs/u30/nedialk/GPU1/code/
matmul_acc.c function=matmul_acc line=13 device=0
num_gangs=16 num_workers=1 vector_length=128 grid=16 block=128
```

This gives

- executable
- file name with accelerator code
- line number of the kernel
- number of gang, workers and vector dimensions
- CUDA grid and block dimensions

PGI_ACC_TIME

To output profiling information, set

```
csh: setenv PGI_ACC_TIME 1
bash: export PGI_ACC_TIME=1
```

Executing `./matmul_acc 4000` gives

```
Accelerator Kernel Timing data
/nfs/u30/nedialk/GPU1/code/mat_mul_acc.c
  matmul_acc  NVIDIA  devicenum=0
    time(us): 470
    10: data region reached 1 time
        10: data copyin transfers: 8
             device time(us): total=313 max=47 min=35 avg=39
        22: data copyout transfers: 4
             device time(us): total=111 max=31 min=26 avg=27
    10: compute region reached 1 time
        13: kernel launched 1 time
            grid: [32]  block: [128]
             device time(us): total=46 max=46 min=46 avg=46
            elapsed time(us): total=58 max=58 min=58 avg=58
```

kernel is launched 1 time, $\approx$ 4.6 seconds