OpenACC. Part I

Ned Nedialkov

McMaster University
Canada

CS/SE 4F03
March 2016

Outline

Execution model

Memory model

CUDA

OpenMP example

OpenACC example

Main program

pgaccelinfo

ACC_NOTIFY

PGI_ACC_TIME

OpenACC

- Set of compiler directives, library routines, and environment variables
- Fortran, C, C++
- Initially developed by PGI, Cray, NVIDIA, CAPS (OpenACC 1.0 in 2011)
- 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
- In this course we use the PGI C compiler, pgcc
- For gcc see `https://gcc.gnu.org/wiki/OpenACC`

References

- OpenACC 2.0 `http://www.openacc.org/sites/default/files/OpenACC%202%200.pdf`
- PGI Accelerator Compilers. OpenACC Getting Started Guide `https://www.pgroup.com/doc/openacc_gs.pdf`
- OpenACC web site `http://www.openacc-standard.org/`
- 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`
- PGI compiler and tools `https://www.pgroup.com/resources/articles.htm`
- 11 Tips for Maximizing Performance with OpenACC Directives in Fortran `https://www.pgroup.com/resources/openacc_tips_fortran.htm`
- David B. Kirk and Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach

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

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

Hardware

- number of streaming multiprocessors (SMs)
- each contains either 8 or 32 (CUDA) cores
- when a kernel is launched thread blocks are distributed to SMs
- threads from a thread block execute concurrently on a single SM
- a SM can execute multiple thread blocks concurrently
- Active thread blocks are managed in groups of warps
  Warp:
  - 32 threads
  - subset of threads from a single block

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
- hard upper limit on the size of a thread block
  Kepler:
  - 1,024 threads or 32 warps
  - SM can have 2,048 threads simultaneously active, or 64 warps

- 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

Mapping

- gang $\equiv$ thread block, worker $\equiv$ warp, vector $\equiv$ threads in warp or
- gang $\equiv$ block, vector $\equiv$ threads in a block

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;
        }
  }
}
```

OpenACC example

```c
#ifdef _OPENACC
#include <openacc.h>
#endif

void matmul_acc(float * restrict C, float * restrict A,
                float * restrict 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 acc kernels copyin(A[0:m*n], B[0:n*p])  copyout(C[0:m*p])
  {
    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;
        }
  }
}
```

Compiling produces

```
pgcc -fast -acc -Minfo -ta=tesla,cc30 -O2   -c -o mat_mul_acc.o mat_mul_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
```

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_mat_mul.o mat_mul_mp.o
        $(CC) $(LDFLAGS) -o $@ $?

clean:
        rm *.o *~ matmul_mp

# makefile for OpenACC
CC=pgcc
CFLAGS=-fast -acc -Minfo -ta=tesla,cc35 -O2
LDFLAGS=-acc -ta=tesla,cc35

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

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

bash scripts

```
#!/bin/bash
# timeall
echo SIZE $1
rm -rf acc$1 omp$1

echo "Timing␣OpenMP␣version"

for ((x=1; x<=16; x*=2))
do
    export OMP_NUM_THREADS=$x
    echo Num threads $x
   ./matmul_mp $1 >> omp$1
done

echo "Timing␣OpenACC␣version"
./matmul_acc $1 >> acc$1

#!/bin/bash
# runall
make clean -f makefile_acc
make -f makefile_acc
make clean
make
./timeall 800
./timeall 2000
./timeall 4000
```

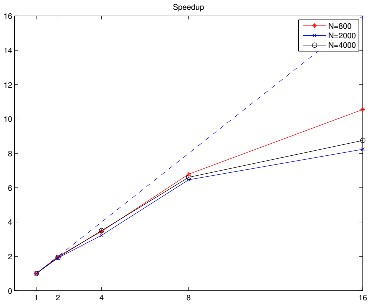
Speedups

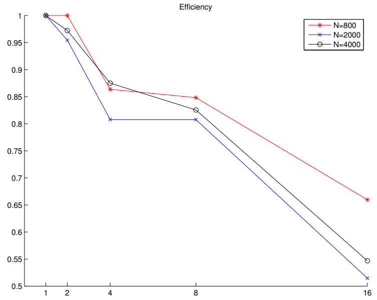
Speedup results on `advol3.mcmaster.ca`

- AMD 8 core Opteron 885, `gcc -O2 -fopenmp`

NVIDIA K40c, Quadro K5000, Quadro K200, with
`pgcc -fast -acc -O2`

	# threads	*N*, seconds/speedup compared to p=1		
		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	4.8/72.9


Speedup
N=800
N=2000
N=4000
0
2
4
6
8
10
12
14
16
1
2
4
8
16


Efficiency
N=800
N=2000
N=4000
0.5
0.55
0.6
0.65
0.7
0.75
0.8
0.85
0.9
0.95
1
1
2
4
8
16

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 31 20:23:19 PDT 2014

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
```

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/
mat_mul_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