An Introduction to OpenACC - Part 1

Feng Chen
HPC User Services
LSU HPC & LONI
sys-help@loni.org

LONI Parallel Programming Workshop
Louisiana State University
Baton Rouge
June 01-03, 2015
Outline of today’s topic

- GPU accelerated computing overview
- OpenACC overview
- First OpenACC program and basic OpenACC directives
- Data region concept
- How to parallelize our examples:
  - Laplacian solver
- Hands-on exercise
  - Matrix Multiplication
  - SAXPY
  - Calculate $\pi$
What is GPU

- A graphics processing unit (GPU) is a computer chip that performs rapid mathematical calculations, primarily for the purpose of rendering images.
- A GPU cluster (Shelob and QB2) /queue is a computer cluster/queue in which each node is equipped with a Graphics Processing Unit (GPU).
GPU Accelerated Computing

- **What is GPU Accelerated Computing**
  - GPU-accelerated computing is the use of a graphics processing unit (GPU) together with a CPU to accelerate scientific, analytics, engineering, consumer, and enterprise applications.

---

**CPU**

**GPU**

---

06/02/2015
GPU Computing History

- The first GPU (Graphics Processing Unit)s were designed as graphics accelerators, supporting only specific fixed-function pipelines.
- Starting in the late 1990s, the hardware became increasingly programmable, culminating in NVIDIA's first GPU in 1999.
- Researchers were tapping its excellent floating point performance. The General Purpose GPU (GPGPU) movement had dawned.
- NVIDIA unveiled CUDA in 2006, the world's first solution for general-computing on GPUs.
- CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model created by NVIDIA and implemented by the GPUs that they produce.
Why is GPU this different from a CPU?

- Different goals produce different designs
  - GPU assumes work load is highly parallel
  - CPU must be good at everything, parallel or not

- CPU architecture must **minimize latency** within each thread
- GPU architecture **hides latency** with computation from other threads
Latency v.s. Throughput

- **F-22 Raptor**
  - 1500 mph
  - Baton Rouge to New York City in 1h 25h
  - Seats 1

- **Boeing 737**
  - 485 mph
  - Baton Rouge to New York City in 4:20h
  - Seats 200
Latency v.s. Throughput

- **F-22 Raptor**
  - Latency
    - 1:25
  - Throughput
    - $1 / 1.42$ hours = 0.7 people/hr.

- **Boeing 737**
  - Latency
    - 4:20
  - Throughput
    - $200 / 4.33$ hours = 46.2 people/hr.
Accelerator Fundamentals

- We must expose enough parallelism to fill the device
  - Accelerator threads are slower than CPU threads
  - Accelerators have orders of magnitude more threads
  - Accelerators tolerate resource latencies by cheaply context switching threads
3 Ways to Accelerate Applications

Applications

Increasing programming effort

Libraries

“Drop-in” Acceleration

CUDA Libraries are interoperable with OpenACC

OpenACC Directives

Easily Accelerate Applications

CUDA Languages are also interoperable with OpenACC

Programming Languages

Maximum Flexibility

An Introduction to OpenACC - Part 1

06/02/2015
Some GPU-accelerated Libraries

- NVIDIA cuBLAS
- NVIDIA cuRAND
- NVIDIA cuSPARSE
- NVIDIA NPP
- GPU VSIPL
- CULA tools
- MAGMA
- NVIDIA cuFFT
- Rogue Wave Software
- IMSL Library
- ArrayFire Matrix Computations
- C++ STL Features for CUDA
- Sparse Linear Algebra
- Vector Signal Image Processing
- GPU Accelerated Linear Algebra
- Matrix Algebra on GPU and Multicore
- CUSP
GPU Programming Languages

**Numerical analytics**
- MATLAB, Mathematica, LabVIEW

**Fortran**
- OpenACC, CUDA Fortran

**C**
- OpenACC, CUDA C

**C++**
- Thrust, CUDA C++

**Python**
- PyCUDA, Copperhead

**F#**
- Alea.cuBase
What is OpenACC

- OpenACC (for Open Accelerators) is a programming standard for parallel computing developed by Cray, CAPS, Nvidia and PGI. The standard is designed to simplify parallel programming of heterogeneous CPU/GPU systems.

- It provides a model for accelerator programming that is portable across operating systems and various types of host CPUs and accelerators.
OpenACC Directives

Your original Fortran or C code

Program myproject
    ... serial code ...
!$acc kernels
    do k = 1,n1
      do i = 1,n2
        ... parallel code ...
      enddo
    enddo
!$acc end kernels
    ...
End Program myproject

Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs
History of OpenACC

- OpenACC was developed by The Portland Group (PGI), Cray, CAPS and NVIDIA. PGI, Cray, and CAPs have spent over 2 years developing and shipping commercial compilers that use directives to enable GPU acceleration as core technology.
- The small differences between their approaches allowed the formation of a group to standardize a single directives approach for accelerators and CPUs.
- Full OpenACC 2.0 Specification available online
  - Implementations available now from PGI, Cray, and CAPS
The Standard for GPU Directives

- **Simple and high-level:**
  - Directive are the easy path to accelerate compute intensive applications. Non-GPU programmers can play along.
  - Single Source: Compile the same program for accelerators or serial, No involvement of OpenCL, CUDA, etc.

- **Open and performance portable:**
  - OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors
  - Supports GPU accelerators and co-processors from multiple vendors, current and future versions.

- **Powerful and Efficient:**
  - Directives allow complete access to the massive parallel power of GPU.
  - Experience shows very favorable comparison to low-level implementations of same algorithms.
  - Developers can port and tune parts of their application as resources and profiling dictates. No need to restructure the program.
Directive-based programming

- Directives provide a high-level alternative
  - Based on original source code (Fortran, C, C++)
  - Easier to maintain/port/extend code
  - Users with OpenMP experience find it a familiar programming model
  - Compiler handles repetitive coding (cudaMalloc, cudaMemcpy...)
  - Compiler handles default scheduling; user tunes only where needed

- Possible performance sacrifice
  - Small performance sacrifice is acceptable
  - Trading-off portability and productivity against this
  - After all, who hand-codes in assembly for CPUs these days?

- As researchers in science and engineering, you often need to balance between:
  - *Time needed to develop your code*
  - *Time needed to focus on the problem itself*
OpenACC Execution Model

- Sequential code executes in a Host (CPU) thread
- Parallel code executes in many Device (GPU) threads across multiple processing elements

**GPU Accelerator**
Optimized for Many Parallel Tasks

Offload to GPU Parallization

**Compute-Intensive Functions**

**Rest of Sequential CPU Code**

CPU
Optimized for Serial Tasks
General Directive Syntax and Scope

- **Fortran**
  
  ```fortran
  !$acc directive [clause [,] clause]...
  
  Often paired with a matching `end` directive surrounding a structured code block
  
  !$acc end directive
  ```

- **C**
  
  ```c
  #pragma acc directive [clause [,] clause]...
  
  { 
  
  Often followed by a structured code block (compound statement) 
  
  }
  ```
The “restrict” keyword in C

- Declaration of intent given by the programmer to the compiler
  - Applied to a pointer, e.g. `float *restrict ptr;`
  - Meaning: “for the lifetime of `ptr`, only it or a value directly derived from it (such as `ptr + 1`) will be used to access the object to which it points”*
  - In simple, the `ptr` will only point to the memory space of itself

- **OpenACC compilers often require restrict to determine independence.**
  - Otherwise the compiler can’t parallelize loops that access `ptr`
  - Note: if programmer violates the declaration, behavior is undefined.

**THE RESTRICT CONTRACT**
I, [insert your name], a PROFESSIONAL or AMATEUR [circle one] programmer, solemnly declare that writes through this pointer will not effect the values read through any other pointer available in the same context which is also declared as restricted.

*Your agreement to this contract is implied by use of the restrict keyword ;)*

The First Simple Exercise: SAXPY

void saxpy(int n, 
    float a, 
    float *x, 
    float *restrict y)
{
    #pragma acc kernels
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);

subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a 
    integer :: n, i
    !$acc kernels
    do i=1,n
        y(i) = a*x(i)+y(i)
    enddo
    !$acc end kernels
end subroutine saxpy

...!

!Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...
Complete saxpy.c

- Only a single line to the above example is needed to produce an OpenACC SAXPY in C.

```c
int main(int argc, char **argv)
{
    int n = 1<<20; // 1 million floats
    float *x = (float*)malloc(n*sizeof(float));
    float *y = (float*)malloc(n*sizeof(float));
    for (int i = 0; i < n; ++i) {
        x[i] = 2.0f;
        y[i] = 1.0f;
    }
    saxpy(n, 3.0f, x, y);
    free(x);
    free(y);
    return 0;
}

void saxpy(int n,
    float a,
    float *x,
    float *restrict y)
{
    #pragma acc kernels
    for (int i = 0; i < n; ++i) {
        y[i] = a*x[i] + y[i];
    }
}
```
SAXPY code (only functions) in CUDA C

// define CUDA kernel function
__global__ void saxpy_kernel( float a, float* x, float* y, int n ){
    int i;
    i = blockIdx.x*blockDim.x + threadIdx.x;
    if( i <= n ) y[i] = a*x[i] + y[i];
}

void saxpy( float a, float* x, float* y, int n ){
    float *xd, *yd;
    // manage device memory
    cudaMalloc( (void**)&xd, n*sizeof(float) );
    cudaMalloc( (void**)&yd, n*sizeof(float) );
    cudaMemcpy( xd, x, n*sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( yd, y, n*sizeof(float), cudaMemcpyHostToDevice );
    // calls the kernel function
    saxpy_kernel<<< (n+31)/32, 32 >>>( a, xd, yd, n );
    cudaMemcpy( x, xd, n*sizeof(float), cudaMemcpyDeviceToHost );
    // free device memory after use
    cudaFree( xd );
    cudaFree( yd );
}
CUDA C/OpenACC – Big Difference

- With CUDA, we changed the structure of the old code. Non-CUDA programmers can’t understand new code. It is not even ANSI standard code.
  - We have separate sections for the host code, and the GPU device code. Different flow of code. Serial path now gone forever.
  - Although CUDA C gives you maximum flexibility, the effort needed for restructuring the code seems to be high.
  - OpenACC seems ideal for researchers in science and engineering.
Add PGI compiler to your environment

Get an interactive session before running your jobs:

```bash
qsub -I -l nodes=1:ppn=16,walltime=4:00:00 -A hpc_train_2015
```

```bash
[hpctrn01@shelob001 ~]$ cat ~/.soft
# This is the .soft file.
# It is used to customize your environment by setting up environment
# variables such as PATH and MANPATH.
# To learn what can be in this file, use 'man softenv'.
+portland-14.3
@default
[hpctrn01@shelob001 ~]$ resoft
[hpctrn01@shelob001 ~]$ pgcc -V
[hpctrn01@shelob001 ~]$ cp -r /home/fchen14/loniworkshop2015/ ./
[hpctrn01@shelob001 ~]$ cd ~/loniworkshop2015/saxpy/openacc/exercise
[hpctrn01@shelob001 ~]$ vi saxpy_1stexample.c
[hpctrn01@shelob001 ~]$ pgcc -acc -Minfo=accel -ta=nvidia,time saxpy_1stexample.c
```

06/02/2015
Compiler output of the first example

- **C**
  
  ```
  pgcc -acc -Minfo=accel -ta=nvidia,time saxpy_1stexample.c
  ```

- **Fortran**
  
  ```
  pgf90 -acc -Minfo=accel -ta=nvidia,time saxpy_1stexample.c
  ```

- **Use “man pgcc/pgf90” to check the meaning of the compiler switches.**

**Compiler output:**

```
pgcc -acc -Minfo=accel -ta=nvidia,time saxpy_1stexample.c
saxpy:
  26, Generating present_or_copyin(x[:n])
  Generating present_or_copy(y[:n])
  Generating NVIDIA code
  27, Loop is parallelizable
  Accelerator kernel generated
  27, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```
Runtime output

[hpctrn01@shelob001 c]$ ./a.out

Accelerator Kernel Timing data
/home/fchen14/loniworkshop2014/ laplace/openacc/c/saxpy_1stexample.c
saxpy  NVIDIA  devicenum=0
time(us): 2,247
26: data region reached 1 time
  26: data copyin reached 2 times
    device time(us): total=1,421 max=720 min=701 avg=710
  29: data copyout reached 1 time
    device time(us): total=637 max=637 min=637 avg=637
26: compute region reached 1 time
  26: kernel launched 1 time
    grid: [4096]  block: [256]
    device time(us): total=189 max=189 min=189 avg=189
    elapsed time(us): total=201 max=201 min=201 avg=201

$2,247 = 1,421 + 637 + 189$
OpenACC kernels directive

- **What is a kernel?** A function that runs in parallel on the GPU.
  - The kernels directive expresses that a region may contain parallelism and the compiler determines what can be safely parallelized.
  - The compiler breaks code in the kernel region into a sequence of kernels for execution on the accelerator device.
  - When a program encounters a `kernels` construct, it will launch a sequence of kernels in order on the device.

- **The compiler identifies 2 parallel loops and generates 2 kernels below.**

```c
#pragma acc kernels
{
  for (i = 0; i < n; i++){
    x[i] = 1.0;
    y[i] = 2.0;
  }
  for (i = 0; i < n; i++){
    y[i] = a*x[i] + y[i];
  }
}

!$acc kernels
do i = 1, n
  x(i) = 1.0
  y(i) = 2.0
end do
do i = 1, n
  y(i) = y(i) + a * x(i)
end do
!$acc end kernels
```
OpenACC parallel directive

- Similar to OpenMP, the parallel directive identifies a block of code as having parallelism.
- Compiler generates one parallel kernel for that loop.
- C
  ```c
  #pragma acc parallel [clauses]
  ```
- Fortran
  ```fortran
  !$acc parallel [clauses]
  ```

```c
#pragma acc parallel
{
  for (i = 0; i < n; i++){
    x[i] = 1.0 ;
    y[i] = 2.0 ;
  }
  for (i = 0; i < n; i++){
    y[i] = a*x[i] + y[i];
  }
}
```

```fortran
!$acc parallel
do i = 1, n
  x(i) = 1.0
  y(i) = 2.0
end do
do i = 1, n
  y(i) = y(i) + a * x(i)
end do
!$acc end parallel
```
Loops are the most likely targets for parallelizing.
- The Loop directive is used within a parallel or kernels directive identifying a loop that can be executed on the accelerator device.
- The loop directive can be combined with the enclosing parallel or kernels
- The loop directive clauses can be used to optimize the code. This however requires knowledge of the accelerator device.
- Clauses: gang, worker, vector, num_gangs, num_workers

**C:**
```c
#pragma acc [kernels/parallel] loop [clauses]
```

**Fortran:**
```fortran
!$acc [kernels/parallel] loop [clauses]
```

```c
#pragma acc loop
for (i = 0; i < n; i++){
    y[i] = a*x[i] + y[i];
}
```

```fortran
!$acc loop
do i = 1, n
    y(i) = y(i) + a * x(i)
end do
!$acc end loop
```
OpenACC kernels vs parallel

- **kernels**
  - Compiler performs parallel analysis and parallelizes what it believes is safe.
  - Can cover larger area of code with single directive.

- **parallel**
  - Requires analysis by programmer to ensure safe parallelism.
  - Straightforward path from OpenMP

- Both approaches are equally valid and can perform equally well.
Clauses

- **data management clauses**
  - `copy(...)`, `copyin(...)`, `copyout(...)`
  - `create(...)`, `present(...)`
  - `present_or_copy{,in,out}(...)` or `pcopy{,in,out}(...)`
  - `present_or_create(...)` or `pcreate(...)`

- **reduction(operator:list)**
- **if (condition)**
- **async (expression)**
Runtime Libraries

- **System setup routines**
  - `acc_init(acc_device_nvidia)`
  - `acc_set_device_type(acc_device_nvidia)`
  - `acc_set_device_num(acc_device_nvidia)`

- **Synchronization routines**
  - `acc_async_wait(int)`
  - `acc_async_wait_all()`

- **For more information, refer to the OpenACC standard**
Second example: Jacobi Iteration

- Solve Laplace equation in 2D:
  - Iteratively converges to correct value (e.g. Temperature), by computing new values at each point from the average of neighboring points.

\[
\nabla^2 f (x, y) = 0
\]

\[
A_{k+1}(i, j) = \frac{A_k(i - 1, j) + A_k(i + 1, j) + A_k(i, j - 1) + A_k(i, j + 1)}{4}
\]
Graphical representation for Jacobi iteration

Current Array: A

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>1.0</td>
<td>2.0</td>
<td>4.0</td>
<td>6.0</td>
<td>8.0</td>
<td>10.0</td>
<td>12.0</td>
<td>1.0</td>
</tr>
<tr>
<td>1.0</td>
<td>3.0</td>
<td>5.0</td>
<td>7.0</td>
<td>9.0</td>
<td>11.0</td>
<td>13.0</td>
<td>1.0</td>
</tr>
<tr>
<td>1.0</td>
<td>2.0</td>
<td>6.0</td>
<td>1.0</td>
<td>3.0</td>
<td>7.0</td>
<td>5.0</td>
<td>1.0</td>
</tr>
<tr>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Next Array: Anew

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>1.0</td>
<td>2.25</td>
<td>3.56</td>
<td>6.0</td>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td>1.0</td>
<td>5.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Serial version of the Jacobi Iteration

```c
while ( error > tol && iter < iter_max )
{
    error=0.0;

    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++) {

            Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1] +
                                  A[j-1][i] + A[j+1][i] );

            error = fmax(error, abs(Anew[j][i] - A[j][i]));
        }
    }

    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
```

Iterate until converged
Iterate across matrix elements
Calculate new value from neighbors
Compute max error for convergence
Swap input/output arrays
First Attempt in OpenACC

// first attempt in C
while ( error > tol && iter < iter_max ) {
    error=0.0;
    #pragma acc kernels
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++) {
            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                                A[j-1][i] + A[j+1][i]);
            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }
    #pragma acc kernels
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
Compiler Output

pgcc -acc -Minfo=accel -ta=nvidia,time laplace_openacc.c -o laplace_acc.out

main:

65, Generating present_or_copyin(Anew[1:4094][1:4094])
Generating present_or_copyin(A[:4096][:4096])
Generating NVIDIA code
66, Loop is parallelizable
67, Loop is parallelizable
   Accelerator kernel generated
   66, #pragma acc loop gang /* blockIdx.y */
   67, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
   70, Max reduction generated for error
75, Generating present_or_copyin(Anew[1:4094][1:4094])
Generating present_or_copyin(A[1:4094][1:4094])
Generating NVIDIA code
76, Loop is parallelizable
77, Loop is parallelizable
   Accelerator kernel generated
   76, #pragma acc loop gang /* blockIdx.y */
   77, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
Performance of First Jacobi ACC Attempt

- CPU: Intel(R) Xeon(R) CPU E5-2670 @ 2.60GHz
- GPU: Nvidia Tesla K20Xm
- The OpenACC code is even slower than the single thread/serial version of the code
- What is the reason for the significant slow-down?

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (sec)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP 1 threads</td>
<td>45.64</td>
<td>--</td>
</tr>
<tr>
<td>OpenMP 2 threads</td>
<td>30.05</td>
<td>1.52</td>
</tr>
<tr>
<td>OpenMP 4 threads</td>
<td>24.91</td>
<td>1.83</td>
</tr>
<tr>
<td>OpenMP 8 threads</td>
<td>25.24</td>
<td>1.81</td>
</tr>
<tr>
<td>OpenMP 16 threads</td>
<td>26.19</td>
<td>1.74</td>
</tr>
<tr>
<td>OpenACC w/GPU</td>
<td>190.32</td>
<td>0.24</td>
</tr>
</tbody>
</table>
Output Timing Information from Profiler

- **Use compiler flag:** `-ta=nvidia, time`
  - Link with a profile library to collect simple timing information for accelerator regions.

- **OR set environmental variable:** `export PGI_ACC_TIME=1`
  - Enables the same lightweight profiler to measure data movement and accelerator kernel execution time and print a summary at the end of program execution.

- **Either way can output profiling information**
## Accelerator Kernel Timing data (1st attempt)

<table>
<thead>
<tr>
<th>Time (us)</th>
<th>Data Region Reached</th>
<th>Kernel Launched</th>
<th>Reduction Kernel Launched</th>
<th>Compute Region Reached</th>
</tr>
</thead>
<tbody>
<tr>
<td>88,460,895</td>
<td>1000 times</td>
<td>1000 times</td>
<td>1000 times</td>
<td>1000 times</td>
</tr>
<tr>
<td>22,281,725</td>
<td>8000 times</td>
<td>2,325,634</td>
<td>25,988</td>
<td>21,905,025</td>
</tr>
<tr>
<td>20,120,805</td>
<td>8000 times</td>
<td>2,334,977</td>
<td>35,063</td>
<td>20,121,342</td>
</tr>
<tr>
<td>2,325</td>
<td>2,320 avg</td>
<td>22,909 max</td>
<td>90 min</td>
<td>21,905,025</td>
</tr>
<tr>
<td>2,334</td>
<td>2,329 avg</td>
<td>2,689 max</td>
<td>99 min</td>
<td>20,121,342</td>
</tr>
<tr>
<td>2,752</td>
<td>2,785 avg</td>
<td>2,496 min</td>
<td>33 avg</td>
<td>16,896,460 min</td>
</tr>
</tbody>
</table>

### Total Time
- **42.4 sec** spent on data transfer
- **42.0 sec** spent on data transfer
- Around **84 sec** on data transfer, huge bottleneck
Overview of the GPU nodes

- **CPU**: Two 2.6 GHz 8-Core Sandy Bridge Xeon 64-bit Processors (16)
  - 64GB 1666MHz Ram

- **GPU**: Two NVIDIA Tesla K20Xm
  - 14 SMX
  - 2688 SP Cores
  - 896 DP Cores
  - 6G global memory

K20Xm Full chip block diagram

SMX (192 SP, 64 DP)
Basic Concepts on Offloading

- CPU and GPU have their respective memory, connected through PCI-e bus

- Processing Flow of the offloading
  1. Copy input data from CPU memory to GPU memory
  2. Load GPU program and execute
  3. Copy results from GPU memory to CPU memory
Excessive Data Transfers

// first attempt in C
while ( error > tol && iter < iter_max ) {
    error=0.0;
#pragma acc kernels
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++) {
            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }
    #pragma acc kernels
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
Rules of Coprocessor (GPU) Programming

- Transfer the data across the PCI-e bus onto the device and keep it there.
- Give the device enough work to do (avoid preparing data).
- Focus on data reuse within the coprocessor(s) to avoid memory bandwidth bottlenecks.
OpenACC Data Management with Data Region

- C syntax
  
  ```c
  #pragma acc data [clause]
  { structured block/statement }
  ```

- Fortran syntax
  
  ```fortran
  !$acc data [clause]
  structured block
  !$acc end data
  ```

- Data regions may be nested.
Data Clauses

- **copy ( list )**
  /* Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region. */

- **copyin ( list )**
  /* Allocates memory on GPU and copies data from host to GPU when entering region. */

- **copyout ( list )**
  /* Allocates memory on GPU and copies data to the host when exiting region. */

- **create ( list )**
  /* Allocates memory on GPU but does not copy. */

- **present ( list )**
  /* Data is already present on GPU from another containing data region. */

- **and present_or_copy[in|out], present_or_create, deviceptr.**
Second Attempt: OpenACC C

```c
#pragma acc data copy(A), create(Anew)

while ( error > tol && iter < iter_max ) {
    error=0.0;

    #pragma acc kernels
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {

            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }
}
```

Copy A in at beginning of loop, out at end. Allocate Anew on accelerator
Second Attempt: OpenACC Fortran

```fortran
!$acc data copy(A), create(Anew)
do while ( err > tol .and. iter < iter_max )
   err=0._fp_kind
!$acc kernels
   do j=1,m
      do i=1,n
         Anew(i,j) = .25_fp_kind * (A(i+1, j ) + A(i-1, j ) + &
                                    A(i , j-1) + A(i , j+1))
      err = max(err, Anew(i,j) - A(i,j))
      end do
   end do
!$acc end kernels
... 
iter = iter +1
end do
!$acc end data
```
Second Attempt: Performance

- Significant speedup after the insertion of the data region directive
- CPU: Intel Xeon CPU E5-2670 @ 2.60GHz
- GPU: Nvidia Tesla K20Xm

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (sec)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP 1 threads</td>
<td>45.64</td>
<td>--</td>
</tr>
<tr>
<td>OpenMP 2 threads</td>
<td>30.05</td>
<td>1.52</td>
</tr>
<tr>
<td>OpenMP 4 threads</td>
<td>24.91</td>
<td>1.83</td>
</tr>
<tr>
<td>OpenACC w/GPU (data region)</td>
<td>4.47</td>
<td>10.21 (serial) 5.57 (4 threads)</td>
</tr>
</tbody>
</table>
Accelerator Kernel Timing data (2nd attempt)

time(us): 4,056,477
54: data region reached 1 time
54: data copyin reached 8 times
  device time(us): total=22,249 max=2,787 min=2,773 avg=2,781
84: data copyout reached 9 times
  device time(us): total=20,082 max=2,510 min=11 avg=2,231
60: compute region reached 1000 times
63: kernel launched 1000 times
  grid: [16x512] block: [32x8]
  device time(us): total=2,314,738 max=2,407 min=2,311 avg=2,314
  elapsed time(us): total=2,323,334 max=2,421 min=2,319 avg=2,323
63: reduction kernel launched 1000 times
  grid: [1] block: [256]
  device time(us): total=24,904 max=78 min=24 avg=24
  elapsed time(us): total=34,206 max=87 min=32 avg=34
71: compute region reached 1000 times
74: kernel launched 1000 times
  grid: [16x512] block: [32x8]
  device time(us): total=1,674,504 max=1,727 min=1,657 avg=1,674
  elapsed time(us): total=1,683,604 max=1,735 min=1,667 avg=1,683

Only 42.2 ms spent on data transfer
Array Shaping

- Compiler sometimes cannot determine size of arrays
  - Sometimes we just need to use a portion of the arrays
  - we will see this example in the exercise
- Under such case, we must specify explicitly using data clauses and array “shape” for this case
- C
  ```
  #pragma acc data copyin(a[0:size]), copyout(b[s/4:3*s/4])
  ```
- Fortran
  ```
  !$pragma acc data copyin(a(1:size)), copyout(b(s/4:3*s/4))
  ```
- The number between brackets are the beginning element followed by the number of elements to copy:
  - [start_element:number_of_elements_to_copy]
  - In C/C++, this means start at `a[0]` and continue for “size” elements.
- Note: data clauses can be used on data, kernels or parallel
Update Construct

- Fortran
  
  #pragma acc update [clause ...]

- C
  
  !$acc update [clause ...]

- Used to update existing data after it has changed in its corresponding copy (e.g. update device copy after host copy changes)

- Move data from GPU to host, or host to GPU. Data movement can be conditional, and asynchronous.
Further Speedups

- OpenACC gives us more detailed control over parallelization via gang, worker, and vector clauses
  - PE (processing element) as a SM (streaming multiprocessor)
  - gang == CUDA threadblock
  - worker == CUDA warp
  - vector == CUDA thread

- By understanding more about OpenACC execution model and GPU hardware organization, we can get higher speedups on this code

- By understanding bottlenecks in the code via profiling, we can reorganize the code for higher performance
Finding Parallelism in your code

- **(Nested) for loops are best for parallelization**
  - Large loop counts needed to offset GPU/memcpy overhead

- **Iterations of loops must be independent of each other**
  - To help compiler:
    - `restrict` keyword
    - `independent` clause

- **Compiler must be able to figure out sizes of data regions**
  - Can use directives to explicitly control sizes

- **Pointer arithmetic should be avoided if possible**
  - Use subscripted arrays, rather than pointer-indexed arrays.

- **Function calls within accelerated region must be inlineable.**
Exercise 1

For the matrix multiplication code
\[ A \cdot B = C \]

where:
\[ a_{i,j} = i + j \]
\[ b_{i,j} = i \cdot j \]
\[ c_{i,j} = \sum_{k} a_{i,k} \cdot b_{k,j} \]

1. For mm_acc_v0.c, speedup the matrix multiplication code segment using OpenACC directives

2. For mm_acc_v1.c:
   - Change A, B and C to dynamic arrays, i.e., the size of the matrix can be specified at runtime;
   - Complete the function matmul_acc using the OpenACC directives;
   - Compare performance with serial and OpenMP results
Exercise 2

- Complete the saxpy example using OpenACC directives.
  \[ \tilde{y} = a \cdot \tilde{x} + \tilde{y} \]

- Calculate the result of a constant times a vector plus a vector:
  - where \( a \) is a constant, \( \tilde{x} \) and \( \tilde{y} \) are one dimensional vectors.

1. Add OpenACC directives for initialization of x and y arrays;
2. Add OpenACC directives for the code for the vector addition;
3. Compare the performance with OpenMP results;
Exercise 3

- Calculate $\pi$ value using the equation:

$$\int_0^1 \frac{4.0}{1.0 + x^2} = \pi$$

with the numerical integration:

$$\sum_{i=1}^{n} \frac{4.0}{1.0 + x_i \cdot x_i} \Delta x \approx \pi$$

1. Complete the code using OpenACC directives