GPU Programming
with CUDA and OpenACC
Axel Koehler - NVIDIA
Heterogeneous Computing

Add GPUs: Accelerate Applications

**CPUs**: designed to run a few tasks quickly.

**GPUs**: designed to run many tasks efficiently.
Minimum Change, Big Speed-up

Application Code

Compute-Intensive Functions

Rest of Sequential CPU Code

GPU

CPU
Ways to Accelerate Applications

Applications

Libraries

Directives (OpenACC)

Programming Languages (CUDA, ..)

CUDA Libraries are interoperable with OpenACC

CUDA Language is interoperable with OpenACC

Easiest Approach

Maximum Performance

No Need for Programming Expertise

High Level Languages (Matlab, ..)

CUDA Libraries are interoperable with OpenACC

CUDA Language is interoperable with OpenACC
## GPU Accelerated Libraries

“Drop-in” Acceleration for Your Applications

<table>
<thead>
<tr>
<th>NVIDIA cuBLAS</th>
<th>NVIDIA cuRAND</th>
<th>NVIDIA cuSPARSE</th>
<th>NVIDIA NPP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GPU VSIPL</strong></td>
<td><strong>CULA tools</strong></td>
<td><strong>MAGMA</strong></td>
<td><strong>NVIDIA cuFFT</strong></td>
</tr>
<tr>
<td>Vector Signal Image Processing</td>
<td>GPU Accelerated Linear Algebra</td>
<td>Matrix Algebra on GPU and Multicore</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>ROGUE WAVE SOFTWARE</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>CUSP</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Sparse Linear Algebra</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>IMSL Library</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>LIBJACET</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Building-block Algorithms for CUDA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Thrust</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>C++ STL Features for CUDA</td>
</tr>
</tbody>
</table>

© NVIDIA Corporation 2012
OpenACC Directives

Easy, Open, Powerful

- Simple Compiler hints
- Works on multicore CPUs & many core GPUs
- Compiler Parallelizes code
- Future Integration into OpenMP standard planned

http://www.openacc.org

Copyright © NVIDIA Corporation 2012

Your original Fortran or C code

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

CPU

GPU

OpenACC Compiler Hint
OpenACC

- Compiler directives to specify parallel regions in C, C++, Fortran
  - OpenACC compilers offload parallel regions from host to accelerator
  - Portable across OSes, host CPUs and accelerators

- Create high-level heterogeneous programs
  - Without explicit accelerator initialization,
  - Without explicit data or program transfers between host and accelerator

- Programming model allows programmers to start simple
  - Enhance with additional guidance for compiler on loop mappings, data location, and other performance details
OpenACC
High-level, with low-level access

Compatible with other GPU languages and libraries
- Interoperate between CUDA C/Fortran and GPU libraries
  - e.g. CUFFT, CUBLAS, CUSPARSE, etc.
OpenACC Directive Syntax

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

- **C**
  ```c
  #pragma acc directive [clause [,] clause] …
  ```
  Often followed by a structured code block
OpenACC Directive Set

- **Kernels Constructs**
  
  #pragma acc kernels [clause ...] new-line

- **Parallel Constructs**
  
  #pragma acc parallel [clause [[,] clause]...] new-line

- **Loop Constructs**
  
  #pragma acc loop [clause [[,] clause]...] new-line

- **Data Constructs**
  
  #pragma acc data [clause [[,] clause]...] new-line

- **Runtime Library Routines**

- **Cache Directives**

- And few others…
kernels: Your first OpenACC Directive

Each loop executed as a separate *kernel* on the GPU.

```c
!$acc kernels
    do i=1,n
        a(i) = 0.0
        b(i) = 1.0
        c(i) = 2.0
    end do

    do i=1,n
        a(i) = b(i) + c(i)
    end do

!$acc end kernels
```

**Kernel:**
A parallel function that runs on the GPU
Example: Jacobi Iteration

- Iteratively converges to correct value (e.g. Temperature), by computing new values at each point from the average of neighboring points.
- Common, useful algorithm
- Example: Solve Laplace equation in 2D: $\nabla^2 f(x, y) = 0$

\[
A_{k+1}(i, j) = \frac{A_k(i-1, j) + A_k(i+1, j) + A_k(i, j-1) + A_k(i, j+1)}{4}
\]
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++) {
            error = max(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++;
}
OpenMP C Code

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

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

iter++;
}
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++;
}
C Makefile (PGI)

CC = pgcc
CCFLAGS =
ACCFLAGS = -acc -ta=nvidia, -Minfo=accel
OMPFLAGS = -fast -mp -Minfo

BIN = laplace2d_omp laplace2d_acc

all: $(BIN)

laplace2d_acc: laplace2d.c
  $(CC) $(CCFLAGS) $(ACCFLAGS) -o $@ $<

laplace2d_omp: laplace2d.c
  $(CC) $(CCFLAGS) $(OMPFLAGS) -o $@ $<

clean:
  $(RM) $(BIN)
Compiler output (PGI)

pgcc  -acc -ta=nvidia -Minfo=accel -o laplace2d_acc laplace2d.c
main:
  57, Generating copyin(A[:4095][:4095])
    Generating copyout(Anew[1:4094][1:4094])
    Generating compute capability 1.3 binary
    Generating compute capability 2.0 binary
  58, Loop is parallelizable
  60, Loop is parallelizable
    Accelerator kernel generated
    58, #pragma acc loop worker, vector(16) /* blockIdx.y threadIdx.y */
    60, #pragma acc loop worker, vector(16) /* blockIdx.x threadIdx.x */
    Cached references to size [18x18] block of 'A'
      CC 1.3 : 17 registers; 2656 shared, 40 constant, 0 local memory bytes; 75% occupancy
      CC 2.0 : 18 registers; 2600 shared, 80 constant, 0 local memory bytes; 100% occupancy
    64, Max reduction generated for error
  69, Generating copyout(A[1:4094][1:4094])
    Generating copyin(Anew[1:4094][1:4094])
    Generating compute capability 1.3 binary
    Generating compute capability 2.0 binary

........
Where time is being spent?

Add `-ta=nvidia,time` to compiler command line

Accelerator Kernel Timing data
`/usr/users/6/harrism/openacc-workshop/solutions/001-laplace2D-kernels/laplace2d.c`

```sh
main 69: region entered 1000 times
  time(us): total=77524918 init=240 region=77524678
  kernels=4422961 data=66464916
/w/o init: total=77524678 max=83398 min=72025 avg=77524
72: kernel launched 1000 times
  grid: [256x256] block: [16x16]
  time(us): total=4422961 max=4543 min=4345 avg=4422
```

```sh
main 57: region entered 1000 times
  time(us): total=82135902 init=216 region=82135686
  kernels=8346306 data=66775717
/w/o init: total=82135686 max=159083 min=76575 avg=82135
60: kernel launched 1000 times
  grid: [256x256] block: [16x16]
  time(us): total=8201000 max=8297 min=8187 avg=8201
  ...
```

Data Transfer Bottleneck!
Computation: 12.7 seconds
Data movement: 133.3 seconds
Basic Concepts

For efficiency, decouple data movement and compute off-load
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]));
        }
    }
}

*A, Anew resident on host

These copies happen every iteration of the outer while loop!*

*Note: there are two #pragma acc kernels, so there are 4 copies per while loop iteration!*
OpenACC C Code (Data Directives)

```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++ ) {

      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++;
}

Copy A in at beginning of loop, out at end. Allocate Anew on accelerator.
Sharing data with libraries

- CUDA libraries and OpenACC both operate on device arrays

- OpenACC provides mechanisms for interop with library calls
  - host_data construct
    - Get device address for pointers inside acc data regions
  - deviceptr data clause
    - Pass pre-allocated device data to OpenACC regions and loops

- The same techniques can be used to share device data between OpenACC loops and
  - Custom CUDA C/C++/Fortran/etc. device code
  - Any CUDA Library that uses CUDA device pointers
<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 1 OpenMP thread</td>
<td>69.80</td>
<td>--</td>
</tr>
<tr>
<td>CPU 2 OpenMP threads</td>
<td>44.76</td>
<td>1.56x</td>
</tr>
<tr>
<td>CPU 4 OpenMP threads</td>
<td>39.59</td>
<td>1.76x</td>
</tr>
<tr>
<td>CPU 6 OpenMP threads</td>
<td>39.71</td>
<td>1.76x</td>
</tr>
<tr>
<td>OpenACC GPU</td>
<td>13.65</td>
<td>2.9x</td>
</tr>
</tbody>
</table>

Note: same code runs in 9.78s on NVIDIA Tesla M2090 GPU
Focus on Exposing Parallelism

With Directives, tuning work focuses on exposing parallelism, which makes codes inherently better.

Example: Application tuning work using directives for new Titan system at ORNL

**S3D**
Research more efficient combustion with next-generation fuels

- Tuning top 3 kernels (90% of runtime)
- 3 to 6x faster on CPU+GPU vs. CPU+CPU
- But also improved all-CPU version by 50%

**CAM-SE**
Answer questions about specific climate change adaptation and mitigation scenarios

- Tuning top key kernel (50% of runtime)
- 6.5x faster on CPU+GPU vs. CPU+CPU
- Improved performance of CPU version by 100%
CUDA 4.1 Highlights

**Advanced Application Development**
- New LLVM-based compiler
- 3D surfaces & cube maps
- Peer-to-Peer between processes
- GPU reset with nvidia-smi
- New GrabCut sample shows interactive foreground extraction
- New code samples for optical flow, volume filtering and more…

**GPU-Accelerated Libraries**
- 1000+ new imaging functions
- Tri-diagonal solver 10x faster vs. MKL
- MRG32k3a & MTGP11213 RNGs
- New Bessell functions in Math lib
- 2x faster matrix-vector w/ HYB-ELL
- Boost-style placeholders in Thrust
- Batched GEMM for small matrices

**New & Improved Developer Tools**
- Re-Designed Visual Profiler
- Parallel Nsight 2.1
- Multi-context debugging
- assert() in device code
- Enhanced CUDA-MEMCHECK
New LLVM-based CUDA Compiler

- Delivers up to 10% faster application performance
- Faster compilation for increased developer productivity
- Modern compiler with broad support
  - Will bring more languages to the GPU
  - Easier to support CUDA to more platforms
NVIDIA Opens Up CUDA Platform

CUDA Compiler Source for Researchers & Tools Vendors

Enables
New Language Support

New Processor Support

Apply for early access at http://developer.nvidia.com/cuda-source
Support GPU to GPU communication through standard MPI interfaces without exposing low level details to the programmer

- e.g. enable MPI_Send, MPI_Recv from/to GPU memory
- Made possible by Unified Virtual Addressing (UVA) in CUDA 4.0
- MVAPICH2, OpenMPI, Platform MPI

**Code without MPI integration**

At Sender:
```
cudaMemcpy(s_buf, s_device, size, cudaMemcpyDeviceToHost);
MPI_Send(s_buf, size, MPI_CHAR, 1, 1, MPI_COMM_WORLD);
```
At Receiver:
```
MPI_Recv(r_buf, size, MPI_CHAR, 0, 1, MPI_COMM_WORLD, &req);
cudaMemcpy(r_device, r_buf, size, cudaMemcpyHostToDevice);
```

**Code with MPI integration**

At Sender:
```
MPI_Send(s_device, size, …);
```
At Receiver:
```
MPI_Recv(r_device, size, …);
```
GPU-aware MVAPICH2

MVAPICH2 (1.8RC1) provides optimized support for GPU to GPU communication through standard MPI interface

- Supports point to point and collective operations
- Pipelined data transfer which *automatically* provides optimizations
- Overlap CUDA copy and RDMA transfer
- Supports GPU Direct (P2P) and CUDA IPC

All possible combinations of source/target available:
- D2D, D2H, H2D
Summary

- Heterogeneous/hybrid Computing is the future
- OpenACC Directives provide a standardized way to hybrid computing
  - Easy, Open and Powerful
- Use highly optimized GPU libraries
- Use CUDA for maximum performance
GPU Technology Conference 2012
May 14-17 | San Jose, CA

The one event you can’t afford to miss

- Learn about leading-edge advances in GPU computing
- Explore the research as well as the commercial applications
- Discover advances in computational visualization
- Take a deep dive into parallel programming

Ways to participate

- Speak - share your work and gain exposure as a thought leader
- Register - learn from the experts and network with your peers
- Exhibit/Sponsor - promote your company as a key player in the GPU ecosystem

www.gputechconf.com
GPU Programming with CUDA and OpenACC
Axel Koehler - NVIDIA (akoehler@nvidia.com)