CUDA & OpenACC

Hyungon Ryu 유지곤 (NVIDIA Korea)

hryu@nvidia.com
Agenda

• Tutorial 1 : GPGPU & CUDA
  – CUDA H/W architecture

• Tutorial 2 : CUDA C/C++ fortran

• Tutorial 3 : CUDA Libraries

• Tutorial 3 : OpenACC

• Tutorial 4 : CUDA 5.0 Kepler Architecture
Tutorial 1
CUDA H/W architecture
Moore’s law is OK....
Conceptual GPU vs. CPU
Conceptual GPU vs. CPU
Conceptual GPU vs. CPU
Conceptual GPU vs. CPU

CPU core

Even load OS kernel

GPU core

Focus on computing
CUDA parallel Model 모델

CPU Program

Kernel Launch

GPU threads
CUDA parallel Model (real)

CPU Program

Kernel Launch

<<< Block, Threads>>>

Block

thread

I-Cache
MT Issue
C-Cache
SP
SP
SP
SP
SP
SP
SFU
SFU
DP
Shared Memory

TPC

SM
Saxpy Example: Serial and OpenMP

saxpy(int n, double a, double*x, double* y) {
    for (int i = 0; i < n; ++i) {
        y[i] = a*x[i] + y[i];
    }
}

saxpy(int n, double a, double*x, double* y) {
    # pragma omp parallel shared (n,a,x,y) private (i)
    # pragma omp for
    for (int i = 0; i < n; ++i) {
        y[i] = a*x[i] + y[i];
    }
}
Saxpy Parallel : CUDA parallel

__global__ void
saxpy(int n, double a, double*x , double* y) {
    int i = blockIdx.x;
    x[i] = a * x[i] + t * y[i];
}

Saxpy <<< n ,1 >>> (n, 2.0, x, y);

<table>
<thead>
<tr>
<th>Loop</th>
<th>configuration</th>
<th>index control</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA : for</td>
<td><strong>global</strong> &amp; &lt;&lt;&lt; A,B&gt;&gt;&gt;&amp;</td>
<td>blockIdx.x</td>
</tr>
<tr>
<td>OpenMP for</td>
<td>#pragma omp for</td>
<td>tid=omp_get_thread_num()</td>
</tr>
</tbody>
</table>

NVIDIA Tutorial
Templates for CUDA for CUDA

#include <stdio.h>

main(){
    Initialize the GPU
    Memory allocation
    Memory copy
    
    FunctionG <<< N , M >>> (parameters);
    Memory copy
    
    Function_host( );
}

void __global__ functionG(parameters){
    functionA( );
    functionB( );
}

__device__ functionA(){
    . . .  }

NVIDIA Tutorial
CUDA C extension

Launch the kernel
    Function <<<< Grid, Block >>>> ( parameter);

Additional C standard API for mem control
    cudaXXX : cudaMemcpy, cudaMemcpy,
    cuXXX    : cudaMemcpy, cudaMemcpy
    cutXXX   : cutXXX

For Function
    __global__, __device__, __host__, __device__ __host__
For memory
    __shared__, __device__, reg/loc

pre-defined variables
    blockDim, blockIdx, threadIdx, cudaMemcpyHostToDevice

Pre-defined function
    __syncthreads(), __mul24(); etc
## Overview of developing

<table>
<thead>
<tr>
<th>Serial</th>
<th>MPI parallel</th>
<th>CUDA parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
<td>Algorithm</td>
<td>Algorithm</td>
</tr>
<tr>
<td>serial Programming</td>
<td>serial Programming</td>
<td>serial Programming</td>
</tr>
<tr>
<td>Compile</td>
<td>Compile</td>
<td>Compile</td>
</tr>
<tr>
<td>Debugging</td>
<td>parallel Programming</td>
<td>CUDA convert</td>
</tr>
<tr>
<td>Release</td>
<td>Profile</td>
<td>Profile</td>
</tr>
<tr>
<td></td>
<td>Parallelize</td>
<td>Parallelize</td>
</tr>
<tr>
<td></td>
<td>Compile</td>
<td>Compile</td>
</tr>
<tr>
<td></td>
<td>Debugging [totalview]</td>
<td>Debugging</td>
</tr>
<tr>
<td></td>
<td>Optimize</td>
<td>Optimize/profile</td>
</tr>
<tr>
<td></td>
<td>Release</td>
<td>Release</td>
</tr>
<tr>
<td></td>
<td>MPIrun</td>
<td></td>
</tr>
</tbody>
</table>
CUDA Parallel programming Steps

1. Check the Algorithm

2. CPU version (serial CPU)

3. OpenMP/MPI parallelization (CPU parallel)

4. CUDA parallel
   A. Global memory control (malloc)
   B. Verify CUDA device functions (static etc)
   C. Parallelize (For Loop to __global__)
   D. Verify Data set (1D, 2D, 3D)
   E. Communication
   F. Verify the results
   G. Optimization

5. done
demo system

OS:
Linux CentOS 6.3

Compiler:
C/C++: GCC + NVCC
fortran: PGI compiler

H/W:
NVIDIA GPU (Tesla C2075)
CUDA C/C++ Model

- **CPU** → **upload** → **GPU**
  - `cudaMemcpy`
  - `GPU_FUNCTION<<<A,B>>>( );`

- **CPU** ← **download** ← **GPU**
  - `cudaMemcpy`
CUDA FORTRAN Model

**upload**

\[ d_{\text{data}} = h_{\text{data}} \]

call GPU_FUNCTION\\(<<<A,B>>>)

**download**

\[ h_{\text{result}} = d_{\text{result}} \]
CUDA C/C++ Makefile Example

#NVCC Makefile Example

include ./Makefile.inc

PROG = saxpy
TARGET = saxpy_gpu
GPU_OBJ = $(PROG).gpu.o
CPU_OBJ = $(PROG).cpu.o
MAIN_OBJ = $(PROG).main.o
OBS = $(GPU_OBJ) $(CPU_OBJ) $(MAIN_OBJ)

all: build
build: $(TARGET)

$(GPU_OBJ): $(PROG)_gpu.c
  @echo "Compile GPU part..."
  @$^ $(NVCC) $(NVCCFLAGS) $(EXTRA_NVCCFLAGS) $(INCLUDES) $(GPU_ARCH) -o $@

$(CPU_OBJ): $(PROG)_cpu.c
  @echo "Compile CPU part..."
  @$^ $(CC) $(CCFLAGS) $(EXTRA_CCFLAGS) $(INCLUDES) -o $@

$(MAIN_OBJ):$(PROG)_main.c
  @echo "Compile main part..."
  @$^ $(CC) $(CCFLAGS) $(EXTRA_CCFLAGS) $(INCLUDES) -o $@

$(TARGET): $(OBS)
  @echo "Link..."
  @$^ $(CC) $(CCFLAGS) $(EXTRA_CCFLAGS) $(INCLUDES) -o $@ $(LDFLAGS) $(EXTRA_LDFLAGS)
un: build
  ./$(TARGET)

clean:
  @rm -rf *.o *.ptx $(TARGET)
  @echo "finish remove all tmp files"
CUDA C/C++ example

```c
__global__ void
saxpy_kernel(float* x, float*y, float alpha, int N){
    int i = blockIdx.x;
    y[i]=alpha*x[i]+ y[i];
    return ;
}

extern "C"
void saxpyGPU(float* x, float* y, float alpha, int N){
    float *x_d, *y_d;
    cudaMalloc( (void**)&x_d  , sizeof(float)*N );
    cudaMalloc( (void**)&y_d  , sizeof(float)*N );

    cudaMemcpy(  x_d , x, sizeof(float)*N ,cudaMemcpyHostToDevice);
    cudaMemcpy(  y_d , y, sizeof(float)*N ,cudaMemcpyHostToDevice);

    saxpy_kernel <<< N,1>>>(x_d, y_d, alpha, N );
    cudaMemcpy(  y , y_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
    return;
}
```
CUDA fortran Makefile example

cat Makefile
#F90FLAGS = -fast
CUDAFLAGS = -Mcuda=cc20

OBJS = increment op_m
all: $(OBJJS)

# increment
increment: increment.cuf op_m.o
    pgfortran $(CUDAFLAGS) $(F90FLAGS) -o $@ $^  
op_m.o: op_m.f
    pgfortran $(CUDAFLAGS) $(F90FLAGS) -c $<

clean:
    rm -rf *.o *.mod $(OBJJS) *~
program cuinfo
  use cudafor

  integer istat, num, numdevices
  type(cudadeviceprop) :: prop

  istat = cudaGetDeviceCount(numdevices)

  do num = 0, numdevices-1
    istat = cudaGetDeviceProperties(prop, num)
    call printDeviceProperties(prop, num)
  end do

end

subroutine printDeviceProperties(prop, num)
  use cudafor
  type(cudadeviceprop) :: prop
  integer num
  ilen = verify(prop%name, ' ', .true.)
  write (*,900) "Device Number: " , num
  write (*,901) "Device Name: " , prop%name(1:ilen)
  write (*,905) "SM: ",prop%maj,prop%min

  900 format (a,i0)
  901 format (a,a)
  905 format (a,i0,'.',i0)

  return
end

check /opt/pgi/linux86-64/2012/src/cudafor.f90
simple CUDA FORTRAN EXAMPLE

program incTest
  use cudafor
  use simpleOps_m
  implicit none
  integer, parameter :: n = 256
  integer :: a(n), b
  integer, device :: a_d(n)

  a = 1
  b = 3

  a_d = a
  call inc<<<1,n>>>(a_d, b)
  a = a_d

  if (all(a == 4)) then
    write(*,*) 'Success'
  endif
end program incTest
cuBLAS / cuFFT
static void simple_dgemm(int n, double alpha, const double *A, const double *B, double beta, double *C)
{
    int i, j, k;
    for (i = 0; i < n; ++i) {
        for (j = 0; j < n; ++j) {
            double prod = 0;
            for (k = 0; k < n; ++k) {
                prod += A[k * n + i] * B[j * n + k];
            }
            C[j * n + i] = alpha * prod + beta * C[j * n + i];
        }
    }
}
cuBLAS Model

CPU

Memcpy

Malloc

GPU

Memcpy

Compute

free

NVIDIA Tutorial
cuBLAS call step

- GPU Memory Alloc
  
cublasAlloc( n_size, memSize, GPU_ptr);

- GPU Memory upload(set)
  
cublasSetVector( n_size, memSize, CPU_ptr, 1, GPU_ptr, 1 );

- GPU Compute
  
cublasSgemm('n', 'n', N, N, N, alpha, d_A, N, d_B, N, beta, d_C, N);

- GPU Memory download(get)
  
cublasGetVector(n2, memSize , GPU_ptr, 1, CPU_ptr, 1);

nvcc mysrc.c –lcuda –lcublas
Main code in cuBLAS

h_A = (float*) malloc(n2 * sizeof(h_A[0]));
h_B = (float*) malloc(n2 * sizeof(h_B[0]));
h_C = (float*) malloc(n2 * sizeof(h_C[0]));

cublasAlloc (n2, sizeof(d_A[0]), (void**)&d_A);
cublasAlloc (n2, sizeof(d_B[0]), (void**)&d_B);
cublasAlloc (n2, sizeof(d_C[0]), (void**)&d_C);

cublasSetVector (n2, sizeof(h_A[0]), h_A, 1, d_A, 1);
cublasSetVector (n2, sizeof(h_B[0]), h_B, 1, d_B, 1);
cublasSetVector (n2, sizeof(h_C[0]), h_C, 1, d_C, 1);

cublasSgemm('n', 'n', N, N, N, alpha, d_A, N, d_B, N, beta, d_C, N);

cublasGetVector(n2, sizeof(h_C[0]), d_C, 1, h_C, 1);
Tutorial 3
OpenACC
OpenACC Specification and Website

- Full OpenACC 1.0 Specification available online
  
  www.openacc.org

- Quick reference card also available

- Beta implementations available now from PGI, Cray, and CAPS
CASE Study

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%

- Tuning top key kernel (50% of runtime)
  - 6.5x faster on CPU+GPU vs. CPU+CPU
  - Improved performance of CPU version by 100%
Familiar to OpenMP Programmers

OpenMP

main() {
  double pi = 0.0; long i;

  #pragma omp parallel for reduction(+:pi)
  for (i=0; i<N; i++)
  {
    double t = (double)((i+0.05)/N);
    pi += 4.0/(1.0+t*t);
  }

  printf("pi = %f\n", pi/N);
}

OpenACC

main() {
  double pi = 0.0; long i;

  #pragma acc kernels
  for (i=0; i<N; i++)
  {
    double t = (double)((i+0.05)/N);
    pi += 4.0/(1.0+t*t);
  }

  printf("pi = %f\n", pi/N);
}
Directive Syntax

• Fortran

  !$acc directive [clause [,] clause]
  ...
  ...often paired with a matching end directive surrounding a structured code block:
  !$acc end directive

• C

  #pragma acc directive [clause [,] clause]
  ...
  ...often followed by a structured code block
kernels: Your first OpenACC Directive

Each loop executed as a separate kernel on the GPU.

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

Fortran

```fortran
!$acc kernels [clause …]
    structured block
!$acc end kernels
```

C

```c
#pragma acc kernels [clause …]
    { structured block }
```

Clauses

```c
if( condition )
async( expression )
```
OpenACC (C/C++) example
MatrixMul
for ( int row=0; row<n; row++ ) {
    for ( int col=0; col<n; col++ ) {
        double val = 0;
        for ( int k=0; k<n; k++ ) {
            val += a[row*n+k] * b[k*n+col];
        }
        c[row*n+col] = val;
    }
}
<table>
<thead>
<tr>
<th>Command</th>
<th>Time</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>./ex1-gcc</code></td>
<td>12.1871 s</td>
<td>0.1762 Gflops/s</td>
</tr>
</tbody>
</table>
CUDA converting

```
for ( int row=0; row<n; row++ ) {
    for ( int col=0; col<n; col++ ) {
        double val = 0;
        for ( int k=0; k<n; k++ ) {
            val += a[row*n+k] * b[k*n+col];
        }
        c[row*n+col] = val;
    }
}
```
NVIDIA Tutorial

OpenACC Converting

```c
#pragma acc data region copyin(a[0:n*n-1],b[0:n*n-1]) copyout(c[0:n*n-1])
{
    #pragma acc region
    {
        #pragma acc for independent
        for ( int row=0; row<n; row++ ) {
            #pragma acc for independent
            for ( int col=0; col<n; col++ ) {
                double val = 0;
                for ( int k=0; k<n; k++ ) {
                    val += a[row*n+k] * b[k*n+col];
                }
                c[row*n+col] = val;
            }
        }
    }
}
```
OpenACC Compile

[cudaguest@gpu03 Ex3]$ make acc


NOTE: your trial license will expire in 13 days, 8.55 hours.
NOTE: your trial license will expire in 13 days, 8.55 hours.
main:
  28, Loop not vectorized: data dependency
  38, Generating copyout(c[0:n*n-1])
    Generating copyin(b[0:n*n-1])
    Generating copyin(a[0:n*n-1])
  43, Generating compute capability 2.0 binary
  47, Loop is parallelizable
  51, Loop is parallelizable
    Accelerator kernel generated
    47, #pragma acc for parallel, vector(16) /* blockIdx.y threadIdx.y */
    51, #pragma acc for parallel, vector(16) /* blockIdx.x threadIdx.x */
    CC 2.0 : 28 registers; 8 shared, 80 constant, 0 local memory bytes
  57, Loop is parallelizable

[cudaguest@gpu03 Ex3]$
### OpenACC result

<table>
<thead>
<tr>
<th>Command</th>
<th>Time</th>
<th>Perf.</th>
</tr>
</thead>
<tbody>
<tr>
<td>./ex3 -cpu</td>
<td>12.1871 s</td>
<td>0.1762 Gflops/s</td>
</tr>
<tr>
<td>./ex3 -acc</td>
<td>0.8621 s</td>
<td>2.4911 Gflops/s</td>
</tr>
</tbody>
</table>
OpenACC(fortran) examples
HEAT
simple Heat Equation

\begin{verbatim}
    wtime = omp_get_wtime ( )

    diff = eps

    ! Iteration start
    do while ( eps <= diff )

        diff = 0.0

        do j = 1, n
            do i = 1, m
                u(i,j) = w(i,j)
            end do
        end do

        do j = 2, n - 1
            do i = 2, m - 1
                w(i,j) = 0.25 * ( u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1) )
            end do
        end do

        do j = 1, n
            do i = 1, m
                diff = max ( diff, abs ( u(i,j) - w(i,j) ) )
            end do
        end do

        iterations = iterations + 1
    end do
\end{verbatim}

sample code form

! Licensing:
! This code is distributed under the GNU LGPL license.
! Modified:
! 18 October 2011
! Author:
! Original FORTRAN90 version by Michael Quinn.
! This version by John Burkardt.

NVIDIA Tutorial
simple Heat Equation

wtime = omp_get_wtime ( )

diff = eps

! Iteration start
  do while ( eps <= diff )

    diff = 0.0

    do j = 1, n
      do i = 1, m
        u(i,j) = w(i,j)
      end do
    end do

    do j = 2, n - 1
      do i = 2, m - 1
        w(i,j) = 0.25 * ( u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1) )
      end do
    end do

    do j = 1, n
      do i = 1, m
        diff = max ( diff, abs ( u(i,j) - w(i,j) ) )
      end do
    end do

    iterations = iterations + 1

  end do
simple Heat Equation with OpenMP

```
do while ( eps <= diff )
   diff = 0.0

!$omp parallel shared ( u, w ) private ( i, j )

   !$omp do
   do j = 1, n
      do i = 1, m
         u(i,j) = w(i,j)
      end do
   end do
!$omp end do

   !$omp do
   do j = 2, n-1
      do i = 2, m-1
         w(i,j) = 0.25 * ( u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1) )
      end do
   end do
!$omp end do

   !$omp do reduction ( max : diff )
   do j = 1, n
      do i = 1, m
         diff = max ( diff, abs ( u(i,j) - w(i,j) ) )
      end do
   end do
!$omp end do

!$omp end parallel
   iterations = iterations + 1
end do
```
simple Heat Equation with OpenACC

do while ( eps <= diff )

    diff = 0.0

!$omp parallel shared ( u, w ) private ( i, j )
!$acc kernels
!$omp do
    do j = 1, n
        do i = 1, m
            u(i,j) = w(i,j)
        end do
    end do
!$omp end do

!$omp do
    do j = 2, n - 1
        do i = 2, m - 1
            w(i,j) = 0.25 * ( u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1) )
        end do
    end do
!$omp end do
!$acc end kernels

!$omp do reduction ( max : diff )
!$acc kernels
    do j = 1, n
        do i = 1, m
            diff = max ( diff, abs ( u(i,j) - w(i,j) ) )
        end do
    end do
!$omp end do
!$acc end kernels

!$omp end parallel
For efficiency, decouple data movement and compute off-load
simple Heat Equation with OpenACC

!$acc data copy(u,w)
do while ( eps <= diff )
diff = 0.0
!$acc kernels
do j = 1, n
   do i = 1, m
      u(i,j) = w(i,j)
   end do
end do

do j = 2, n - 1
   do i = 2, m - 1
      w(i,j) = 0.25 * ( u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1) )
   end do
end do
!$acc end kernels

!$acc kernels
   do j = 1, n
      do i = 1, m
         diff = max ( diff, abs ( u(i,j) - w(i,j) ) )
      end do
   end do
!$omp end do
!$acc end kernels

iterations = iterations + 1

end do
!$acc end data
Kepler Architecture and CUDA5.0
Tesla CUDA Architecture Roadmap

DP GFLOPS per Watt

- **Tesla**
  - 2008
- **Fermi**
  - 2010
- **Kepler**
  - 2012
- **Maxwell**
  - 2014

NVIDIA Tutorial
Graph showing the comparison of Peak Double Precision FP (GFlops/sec) and Peak Memory Bandwidth (GBytes/sec) between NVIDIA GPU, x86 CPU, and specific models over the years 2007 to 2012.
Kepler GK110 Block Diagram

Architecture

- 7.1B Transistors
- 15 SMX units
- > 1 TFLOP FP64
- 1.5 MB L2 Cache
- 384-bit GDDR5
- PCI Express Gen3
Power vs Clock Speed Example

<table>
<thead>
<tr>
<th></th>
<th>Logic</th>
<th>Clocking</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Area</td>
<td>Power</td>
</tr>
<tr>
<td>Fermi</td>
<td>1.0x</td>
<td>1.0x</td>
</tr>
<tr>
<td>Kepler</td>
<td>1.8x</td>
<td>0.9x</td>
</tr>
</tbody>
</table>

NVIDIA Tutorial
## SMX Balance of Resources

<table>
<thead>
<tr>
<th>Resource</th>
<th>Kepler GK110 vs Fermi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating point throughput</td>
<td>2-3x</td>
</tr>
<tr>
<td>Max Blocks per SMX</td>
<td>2x</td>
</tr>
<tr>
<td>Max Threads per SMX</td>
<td>1.3x</td>
</tr>
<tr>
<td>Register File Bandwidth</td>
<td>2x</td>
</tr>
<tr>
<td>Register File Capacity</td>
<td>2x</td>
</tr>
<tr>
<td>Shared Memory Bandwidth</td>
<td>2x</td>
</tr>
<tr>
<td>Shared Memory Capacity</td>
<td>1x</td>
</tr>
</tbody>
</table>
New Instruction: SHFL

Data exchange between threads within a warp

- Avoids use of shared memory
- One 32-bit value per exchange
- 4 variants:

```
Indexed
any-to-any

Shift right to n\textsuperscript{th} neighbour

Shift left to n\textsuperscript{th} neighbour

Butterfly (XOR) exchange
```
Dynamic Parallelism

The ability to launch new grids from the GPU
– Dynamically
– Simultaneously
– Independently

Fermi: Only CPU can generate GPU work
Kepler: GPU can generate work for itself
Dynamic Parallelism

GPU as Co-Processor

Autonomous, Dynamic Parallelism
NVIDIA Tutorial
Better Programming Model

LU decomposition (Fermi)

dgetrf(N, N) {
    for j=1 to N
        for i=1 to 64
            idamax<<<>>>(i)
            memcpy
            dswap<<<>>>(i)
            memcpy
            dscal<<<>>>(i)
            dger<<<>>>(i)
        next i
    memcpy
    dlaswap<<<>>>(i)
    dtrsm<<<>>>(i)
    dgemm<<<>>>(i)
} next j

LU decomposition (Kepler)

dgetrf(N, N) {
    dgetrf<<<>>>(i)
    synchronize();
}
Concurrency: Fermi

Fermi allows 16-way concurrency

- Up to 16 grids can run at once
- But CUDA streams multiplex into a single queue
- Overlap only at stream edges
Concurrency: Kepler

Kepler allows 32-way concurrency

- One work queue per stream
- Concurrency at full-stream level
- No inter-stream dependencies
Hyper-Q: Fermi without Hyper-Q

GPU Utilization %

A  B  C  D  E  F
감사합니다.