Introduction to GPU programming with OpenACC

Research Computing Bootcamp
November 1st, 2019

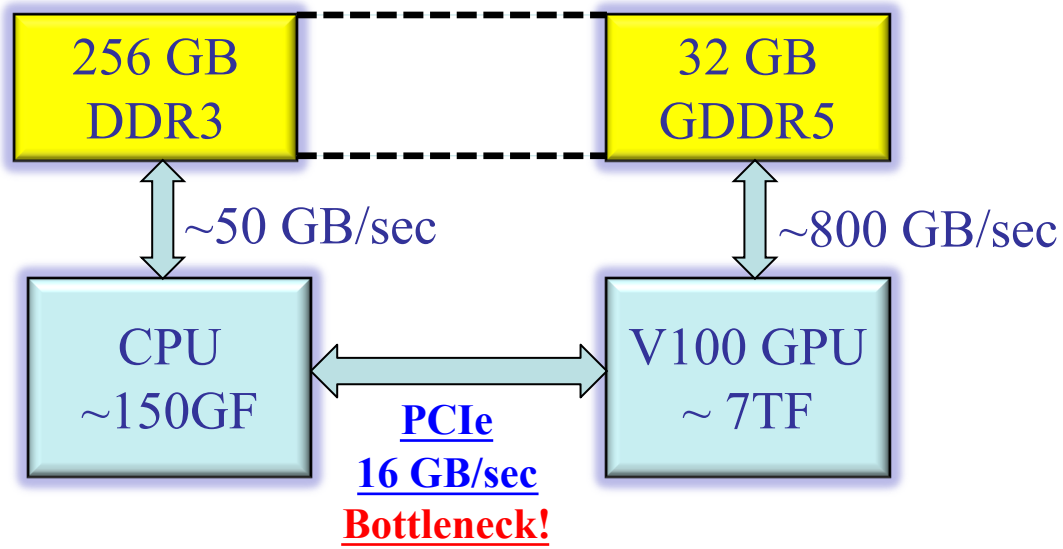
Stéphane Ethier
(ethier@pppl.gov)
Princeton Plasma Physics Laboratory

Slides: ***http://w3.pppl.gov/~ethier/ PICSCIE/ Intro_to_OpenACC_Nov_2019.pdf***

How to program GPUs

- Several possibilities
  - **CUDA**: NVIDIA-specific programming language built as an extension of standard C language. Best approach to get the most out of your GPU. CUDA kernels not portable though. Also available for FORTRAN but only through the PGI compiler.
  - **OpenACC** compiler directives similar to OpenMP. Portable code. Easy to get started. Available for a few compilers. Now can also run on CPU!
  - **OpenMP** via the "target" directive. Not quite as full-featured as OpenACC but getting there
  - Libraries, commercial software, domain-specific environments, . . .
  - OpenCL: open standard, platform- and vendor independent
    - Works on both GPU AND CPU.
    - Very complex. Even harder than CUDA…
    - Very small user base (low adoption)

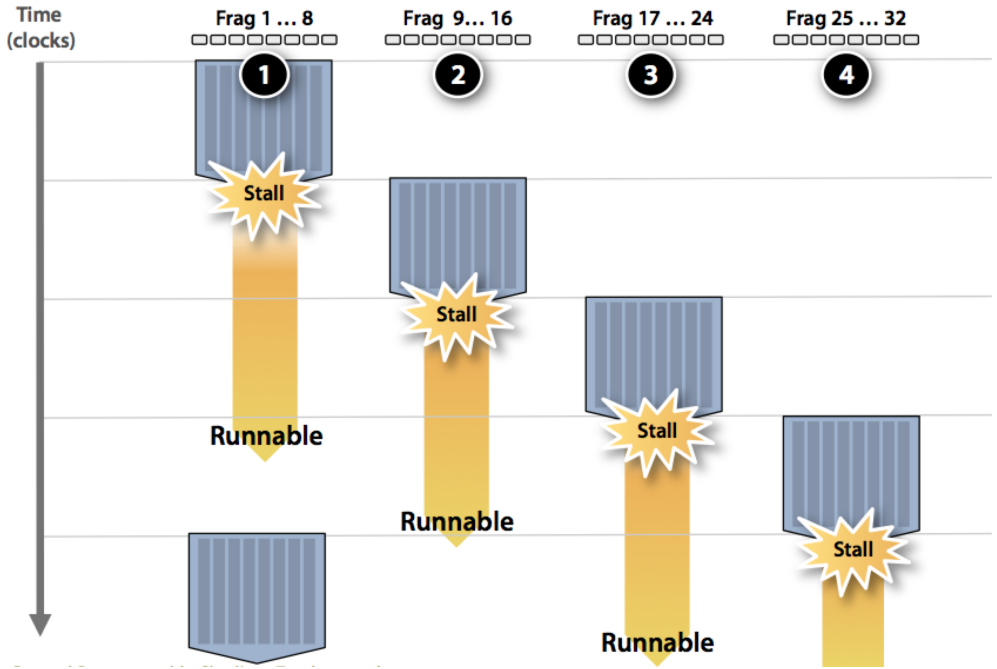
Hardware considerations affecting GPU programming



Nvidia NVLINK on IBM Power 9 system = 75 GB/sec (150 GB/s both ways)

- Keep "kernel" data resident on GPU memory as much as possible
- Avoid frequent copying between CPU and GPU
- Use asynchronous, non-blocking, communication, multi-level overlapping

The secret to GPU high throughput:
massive multi-threading + interleaving



NOT SIMD but rather

SIMT!
Single Instruction Multiple Threads

255 registers per thread!!

http://s08.idav.ucdavis.edu/fatahalian-gpu-architecture.pdf

What to do first…

- MOST IMPORTANT:
  - Find the most time-consuming sections of your code

    HOW? Use Profiler tool!! (ARM MAP for example)
  - Find and expose as much parallelism as you can in your code
  - You need LOTS of parallel operations to keep the GPU busy!
  - Try to remove as many dependencies as you can between successive iterations in a loop.
  - The ideal case is when each iteration is completely independent from the others ➔ VECTORIZATION

Compiling an OpenACC code with PGI

- **`pgcc -acc -Minfo=all -Mneginfo`** (same for pgcc and pgCC), or **`-Minfo=accel`**
  - **`-Minfo=all`** outputs messages from the compiler about optimization, parallelization, etc.
  - **`-Mneginfo`** Outputs messages about why a section was not vectorized or parallelized
- Add target hardware: **`-ta=tesla:cc70`** **(for V100)**
  - Use "pgaccelinfo" on GPU node to find above CUDA version (7.0 here for Tesla Volta V100, 3.5 for K40c)

Resources at Princeton

- Tiger computer
  - https://www.princeton.edu/researchcomputing/computational-hardware/tiger/
  - Nvidia Tesla “Pascal” P100 GPUs!
  - Access to GPU partition via SLURM (#SBATCH –gres=gpu:4)
  - https://www.princeton.edu/researchcomputing/education/online-tutorials/getting-started/

- Adroit
  - 1 node with 4 x V100 Volta GPU (newer than Pascal) and 1 node with 2 K40c
- Use SLURM command `scontrol show node` to view the hardware on the cluster

OpenACC

- http://www.openacc.org
- ***Directive-based*** programming model to direct the compiler in generating GPU-specific instructions
- Least changes to your code
- It is portable across different platforms and compilers
- Not all compilers support OpenACC though
  - PGI, CRAY, CAPS, GCC-6, although **PGI is the best**
- With PGI, the same OpenACC code can run in parallel on both multi-core CPUs and GPUs! (OpenMP trying to do the same…)
- Hides a lot of the complexity
- Works for Fortran, C, C++
- Not as much control over the GPU hardware though. To extract the last bit of performance, CUDA probably a better choice

What are directives?

- In C/C++, preprocessor statements ARE directives. They "direct" the preprocessing stage.
- Parallelization directives tell the compiler to add some machine code so that the next set of instructions will be distributed to several processors and run in parallel.
- In FORTRAN, directives are special purpose comments
- In C/C++, "pragmas" are used to include special purpose directives

C:

Can also be "kernels"

```
#pragma acc parallel loop
for (idx=1; idx <= n; idx++) {
   a[idx] = b[idx] + c[idx];
}
```

Fortran:

```
!$acc parallel loop
do idx=1,n
   a(idx) = b(idx) + c(idx)
enddo
```

Example of OpenACC directive in Fortran

It can be as simple as the following:

```
subroutine smooth( a, b, w0, w1, w2, n, m)
 real, dimension(:,:) :: a,b
 real :: w0, w1, w2
 integer :: n, m
 integer :: i, j
!$acc parallel loop
   do i = 2,n-1
    do j = 2,m-1
     a(i,j)= w0 * b(i,j) + &
        w1 * (b(i-1,j) + b(i,j-1) + b(i+1,j) + b(i,j+1)) + &
        w2 * (b(i-1,j-1) + b(i-1,j+1) + b(i+1,j-1) + b(i+1,j+1))
    enddo
   enddo
```

Accelerator compute constructs (2 possibilities)

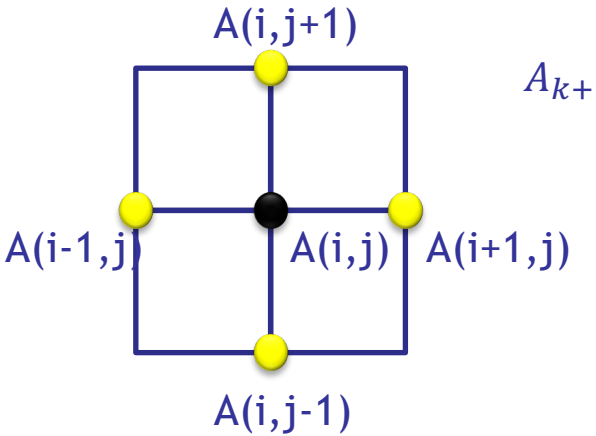
```
#pragma acc parallel [clause-list]  newline
    { structured block, almost always a loop}

!$acc parallel [clause-list]
   structured block,   !$acc loop
!$acc end parallel

#pragma acc kernels [clause-list] newline
   { structured block}

!$acc kernels [clause-list]
    structured block
!$acc end kernels
```

Parallel construct is more **explicit** and gives the **programmer** more responsibility on how the work will be divided between gangs, workers, and vector.

Kernels construct is more **implicit**. It relies on the **compiler** to divide the work by creating an unspecified number of kernels to run on the GPU. Good place to start for beginners!

Exercise code: 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: $\boldsymbol{\nabla^2 f(x, y) = 0}$



$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

Exercise code: Jacobi iteration

```
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 = 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++;
}
```

Iterate until converged

Iterate across matrix elements

Calculate new value from neighbors

Compute max error for convergence

Swap input/output arrays

Exercise #1

- Log onto adroit: `ssh -Y adroit.princeton.edu`
- Load the PGI compiler module:
  - `module purge`
  - `module load pgi`
- Copy the following directory in your home directory:
  - `cp -r /home/ethier/Fall_Bootcamp_2019/OPENACC .`
- Pick C or Fortran and "cd" into corresponding directory:
  - `cd OPENACC/C  (or Fortran)`
- Build and run the code:
  - `make exercise` (have a look at the compiler messages)
  - `sbatch slurm_script`

Exercise #1 continued

- Add **`acc parallel loop`** at the proper locations
- Run again…
- Did it work? How do the timings compare?

- Set the environment variable PGI_ACC_NOTIFY to 3 to get useful information and run again
  - **`export PGI_ACC_NOTIFY=3`**
- What do you see?
- Try using **`acc kernels`** instead of **`parallel loop`**. Is there a difference?
- Let's use the Nvidia profile **`nvprof`** and its graphical interface **`nvvp`**

Hmm... I thought that GPUs were fast

```
PROGRAM main
 INTEGER :: a(N)
 <stuff>
!$acc parallel loop
  DO i = 1,N
     a(i) = I
  ENDDO
!$acc end parallel loop
!$acc parallel loop
  DO i = 1,N
     a(i) = 2*a(i)
  ENDDO
!$acc end parallel loop
<stuff>
END PROGRAM main
```

- Two accelerator parallel region
- Compiler creates two kernels
  - Loop iterations automatically divided across gangs, workers, vectors
  - Breaking parallel regions acts as a barrier
- First kernel initializes array
  - Compiler will determine **copyout(a)**
- Second kernel updates array
  - Compiler will determine **copy(a)** (in and out)
- **Array a(:) unnecessarily moved from and to GPU between kernels**
  - **"data sloshing"**

Much improved version…

```
PROGRAM main
 INTEGER :: a(N)
 <stuff>
!$acc data copyout(a)
!$acc parallel loop
  DO i = 1,N
     a(i) = I
  ENDDO
!$acc end parallel loop
!$acc parallel loop
  DO i = 1,N
     a(i) = 2*a(i)
  ENDDO
!$acc end parallel loop
!$acc end data
<stuff>
END PROGRAM main
```

- Now added a data region
- Specified arrays only moved at boundaries of data region
- Unspecified arrays moved by each kernel
- No compiler-determined movements for data regions
- Data region can contain host code and accelerator regions
- Host and device arrays are independent of each other
- **No automatic synchronization of copies within data region**
  - **User-directed synchronization via update directive**

Data clauses

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.
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. (Structured Only)
create (*list*)	Allocates memory on GPU but does not copy.
delete(*list*)	Deallocate memory on the GPU without copying. (Unstructured Only)
present (*list*)	Data is already present on GPU from another containing data region.

Pay special attention to data structures if working with older versions of PGI compiler

- PGI <18.7 DOES NOT support “deep copy” of data structures.
- Both in C/C++ and Fortran, it is not sufficient to **copy** the pointer to the structure to GPU
- Copy the pointer first
- Then copy each element (vectors, arrays, etc.) explicitly (deep copy)

```
#pragma acc data copyin(A) \
copyin(A.row_offsets[:num_rows+1],A.cols[:nnz],A.coefs[:nnz])
```

Exercise #2

- Go back to the Jacobi example and add some data clauses to improve data movement
- Which ones do you need?
- Recompile, run, and profile again

Example of OpenACC vs CUDA

- Simple example: **REDUCTION** (4 lines of Fortran)

```
a=0.0
do i = 1,n
   a = a + b(i)
end do
```

Reduction in "simple" CUDA

```
__global__ void reduce0(int *g_idata, int *g_odata)
{
extern __shared__ int sdata[];

unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();

for(unsigned int s=1; s < blockDim.x; s *= 2) {
if ((tid % (2*s)) == 0) {
sdata[tid] += sdata[tid + s];
}
__syncthreads();
}

if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}


extern "C" void reduce0_cuda_(int *n, int *a, int *b)
{
int *b_d,red;
const int b_size = *n;


cudaMalloc((void **) &b_d , sizeof(int)*b_size);
cudaMemcpy(b_d, b, sizeof(int)*b_size,
cudaMemcpyHostToDevice);
```

```
dim3 dimBlock(128, 1, 1);
dim3 dimGrid(2048, 1, 1);
dim3 small_dimGrid(16, 1, 1);

int smemSize = 128 * sizeof(int);
int *buffer_d, *red_d;
int *small_buffer_d;

cudaMalloc((void **) &buffer_d , sizeof(int)*2048);
cudaMalloc((void **) &small_buffer_d ,
sizeof(int)*16);
cudaMalloc((void **) &red_d , sizeof(int));


reduce0<<< dimGrid, dimBlock, smemSize >>>(b_d,
buffer_d);

reduce0<<< small_dimGrid, dimBlock, smemSize
>>>(buffer_d, small_buffer_d);

reduce0<<< 1, 16, smemSize >>>(small_buffer_d,
red_d);

cudaMemcpy(&red, red_d, sizeof(int),
cudaMemcpyDeviceToHost);

*a = red;

cudaFree(buffer_d);
cudaFree(small_buffer_d);
cudaFree(b_d);
}
```

Slower than OpenACC Version!!

Ref: SC13 OpenACC tutorial, Luiz DeRose, Alistair Hart, Heidi Poxon, & James Beyer

Reduction code in optimized CUDA

```
template<class T>
struct SharedMemory
{
    __device__ inline operator       T*()
    {
        extern __shared__ int __smem[];
        return (T*)__smem;
    }

    __device__ inline operator const T*() const
    {
        extern __shared__ int __smem[];
        return (T*)__smem;
    }
};

template <class T, unsigned int blockSize, bool nIsPow2>
__global__ void
reduce6(T *g_idata, T *g_odata, unsigned int n)
{
    T *sdata = SharedMemory<T>();

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockSize*2 + threadIdx.x;
    unsigned int gridSize = blockSize*2*gridDim.x;

    T mySum = 0;
    while (i < n)
    {
        mySum += g_idata[i];
        if (nIsPow2 || i + blockSize < n)
            mySum += g_idata[i+blockSize];
        i += gridSize;
    }
sdata[tid] = mySum;
    __syncthreads();

    if (blockSize >= 512) { if (tid < 256) { sdata[tid] = mySum = mySum
+ sdata[tid + 256]; } __syncthreads(); }
    if (blockSize >= 256) { if (tid < 128) { sdata[tid] = mySum = mySum
+ sdata[tid + 128]; } __syncthreads(); }
    if (blockSize >= 128) { if (tid <  64) { sdata[tid] = mySum = mySum
+ sdata[tid +  64]; } __syncthreads(); }
```

```
    if (tid < 32)
    {
        volatile T* smem = sdata;
        if (blockSize >=  64) { smem[tid] = mySum = mySum + smem[tid + 32];  }
        if (blockSize >=  32) { smem[tid] = mySum = mySum + smem[tid + 16];  }
        if (blockSize >=  16) { smem[tid] = mySum = mySum + smem[tid + 8];  }
        if (blockSize >=   8) { smem[tid] = mySum = mySum + smem[tid + 4];  }
        if (blockSize >=   4) { smem[tid] = mySum = mySum + smem[tid + 2];  }
        if (blockSize >=   2) { smem[tid] = mySum = mySum + smem[tid + 1];  }
    }

    if (tid == 0)
        g_odata[blockIdx.x] = sdata[0];
}
extern "C" void reduce6_cuda_(int *n, int *a, int *b)
{
    int *b_d;
    const int b_size = *n;

    cudaMalloc((void **) &b_d , sizeof(int)*b_size);
    cudaMemcpy(b_d, b, sizeof(int)*b_size, cudaMemcpyHostToDevice);

    dim3 dimBlock(128, 1, 1);
    dim3 dimGrid(128, 1, 1);
    dim3 small_dimGrid(1, 1, 1);
    int smemSize = 128 * sizeof(int);
    int *buffer_d;
    int small_buffer[4],*small_buffer_d;

    cudaMalloc((void **) &buffer_d , sizeof(int)*128);
    cudaMalloc((void **) &small_buffer_d , sizeof(int));
    reduce6<int,128,false><<< dimGrid, dimBlock, smemSize >>>(b_d,buffer_d, b_size);
    reduce6<int,128,false><<< small_dimGrid, dimBlock, smemSize
>>>(buffer_d, small_buffer_d,128);
    cudaMemcpy(small_buffer, small_buffer_d, sizeof(int),
cudaMemcpyDeviceToHost);

    *a = *small_buffer;

    cudaFree(buffer_d);
    cudaFree(small_buffer_d);
    cudaFree(b_d);
}
```

Ref: SC13 OpenACC tutorial, Luiz DeRose, Alistair Hart, Heidi Poxon, & James Beyer

OpenACC version of the Reduction code
(10 lines → the compiler does the rest)

```
!$acc data present(a,b,n)
   a = 0.0
!$acc update device(a)
!$acc parallel
!$acc loop reduction(+:a)
   do i = 1,n
      a = a + b(i)
   end do
!$acc end parallel
!$acc end data
```

start data region. a,b,n already in GPU memory

"a" is set to zero on the host (CPU) but not on the "device" (GPU)

host changed the value of "a" so update GPU "a"

start code region (kernel) that will run on GPU

split loop between threads, reduction on "a"

end of kernel region

end of data region

Complete OpenACC specification

http://www.openacc.org/specification

And Programming guide:

http://www.openacc.org/sites/default/files/OpenACC_Programming_Guide_0.pdf

Extremely useful online resources

- OpenACC resources
  - www.openacc.org/resources (tutorials, videos, guides, …)
- Nvidia courses and tutorials
  - https://developer.nvidia.com/accelerated-computing-training
  - https://developer.nvidia.com/openacc-courses
    - Watch the courses
    - Look for link to "OpenACC Toolkit Download" at the bottom
    - Sign up for "Qwiklabs": https://developer.nvidia.com/qwiklabs-signup
- PGI compiler http://www.pgroup.com/
  - Check documentation for compiler, PGPROF, and OpenACC acceleration at http://www.pgroup.com/resources/accel.htm