Multi GPU programming

Alexey A. Romanenko
arom@ccfit.nsu.ru
Novosibirsk State University
Why multi-GPU?

- To further speedup computation
- Working set exceeds a single GPU’s memory
- Having multiple GPUs per node improves perf/W
  - Amortize the CPU server cost among more GPUs
  - Same goes for the price
Hybrid systems

Data by Paulius Micikevicius, NVIDIA
Notes on NUMA architecture

- CPU NUMA affects PCIe transfer throughput in dual-IOH systems
  - Transfers to “remote” GPUs achieve lower throughput
    - One additional QPI hop
  - This affects any PCIe device, not just GPUs
    - Network cards, for example
  - When possible, lock CPU threads to a socket that’s “closest” to the GPU
    - For example, by using numactl, GOMP_CPU_AFFINITY, KMP_AFFINITY, etc.
- Number of hops slightly effect on data transfer throughput
Local H2D Copy: 5.7 GB/s
Remote H2D Copy: 4.9 GB/s
Local D2H Copy: 6.3 GB/s
Remote D2H Copy: 4.9 GB/s
int deviceCount;
cudaGetDeviceCount(&deviceCount);
int device;
for (device = 0; device < deviceCount; ++device) {
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, device);
    printf("Device %d has compute capability %d.%d.\n",
            device, deviceProp.major, deviceProp.minor);
}
CUDA context

- **CUDA context** – device-specific runtime configuration info (allocated device-memory, error codes, etc.)
- Many CUDA calls require existing context
- Initially thread/process does not have current CUDA context
- If thread/process does not have CUDA context, but it is required, then it will be created implicitly
- One device can have multiple contexts (driver API)
Context management

* CUDA runtime API:
  * Context is created implicitly
  * Switching context: `cudaSetDevice(<device number>)`
* Driver API:
  * `cuCtxCreate/cuCtxDestroy`
  * `cuCtxPushCurrent/cuCtxPopCurrent`
• CPU thread assigned with one GPU *
• GPU could be selected explicitly (cudaSetDevice()) or implicitly – by default.
• By default GPU with index «0» is selected
• cudaSetDevice should be the first CUDA related call.

* - false for driver level.
GPU devices and CPU thread
CUDA 4.0

* Any CPU thread can communicate with any GPU
* cudaSetDevice() function select active GPU
* Parallel kernel execution from different CPU threads is possible.
Multi-thread/parallel programming

- OpenMP
- POSIX Threads
- WinThreads
- MPI
- IPC
- etc.
Copying data between GPUs

**CUDA 3.2**
cudaMemcpy(Host, GPU1);
cudaMemcpy(GPU2, Host);

**CUDA 4.0**
cudaMemcpy(GPU1, GPU2);

Tesla 20xx (Fermi) required
64-bit applications
Unified Virtual Addressing
CUDA 4.0

- CPU and GPU allocations use unified virtual address space.
- Driver/device can determine from an address where data resides.
- A given allocation still resides on a single device (an array doesn’t span several GPUs).
- One parameter (cudaMemcpyDefault) instead of 4 (cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice).
- Requires:
  - 64-bit Linux or 64-bit Windows with TCC driver.
  - Fermi or later architecture GPUs (compute capability 2.0 or higher).
  - CUDA 4.0 or later.
Two interesting aspects:

- Peer-to-peer (P2P) memcopies
- Accessing another GPU’s addresses

Both require peer-access to be enabled:

- `cudaDeviceEnablePeerAccess( peer_device, 0 )`
  - Enables current GPU to access addresses on `peer_device` GPU
- `cudaDeviceCanAccessPeer( &accessible, dev_X, dev_Y )`
  - Checks whether `dev_X` can access memory of `dev_Y`
  - Returns 0/1 via the first argument

Peer-access is not available if:

- One of the GPUs is pre-Fermi
- GPUs are connected to different Intel IOH chips on the motherboard
Peer-to-peer memcopy

- cudaMemcpyPeerAsync (void* dst_addr, int dst_dev,
  void* src_addr, int src_dev,
  size_t num_bytes, cudaStream_t stream)
  - Copies the bytes between two devices
  - Currently performance is maximized when stream belongs to the source GPU
  - There is also a blocking (as opposed to Async) version
- If peer-access is enabled:
  - Bytes are transferred along the shortest PCIe path
  - No staging through CPU memory
- If peer-access is not available
  - CUDA driver stages the transfer via CPU memory
How Does P2P Memcopy Help Multi-GPU?

- Ease of programming
  - No need to manually maintain memory buffers on the host for inter-GPU exchanges
- Increased throughput
  - Especially when communication path does not include IOH (GPUs connected to a PCIe switch):
    - Single-directional transfers achieve up to \(~6.6\) GB/s
    - Duplex transfers achieve \(~12.2\) GB/s
    - 4-5 GB/s if going through the host
  - GPU-pairs can communicate concurrently if paths don’t overlap
This section briefly describe approaches to developing of parallel programs.
OpenMP

- Implementation – directives (extension of C, Fortran, ...), library
- Runtime-library is responsible for thread creation/completion, user can specify threads properties explicitly.
- User can manage threads interaction.
OpenMP

* Parallel execution
  `#pragma omp parallel`
* Number of CPU threads
* `omp_get_num_threads()`, `OMP_NUM_THREADS`
* Parallel loops
  `#pragma omp parallel for`
* Parallel sections
  `#pragma omp sections`
* #pragma omp parallel sections
  {
    #pragma omp section
    {
      cudaSetDevice(0);
      ...
    }
    #pragma omp section
    {
      cudaSetDevice(1);
      ...
    }
  }
* #pragma omp parallel sections
  {
    #pragma omp section
    {
      // section for GPUs
      ...
    }
    #pragma omp section
    {
      // section for CPUs
      ...
    }
  }
int nElem = 1024;
cudaGetDeviceCount(&nGPUs);
if(nGPUs >= 1){
    omp_set_num_threads(nGPUs);

#pragma omp parallel
{
    unsigned int cpu_thread_id = omp_get_thread_num();
    unsigned int num_cpu_threads = omp_get_num_threads();
    cudaSetDevice(cpu_thread_id % nGPUs); //set device

    dim3 BS(128);
    dim3 GS(nElem / (gpu_threads.x * num_cpu_threads));
    // memory allocation and initialization
    int startIdx = cpu_thread_id * nElem / num_cpu_threads;
    int threadNum = nElem / num_cpu_threads;
    kernelAddConstant<<<GS, BS>>>(pData, startIdx, threadNum); // memory copying
}

// memory copying
// Section for GPUs.
#pragma omp section
{
#pragma omp parallel for
  for (int i = 0; i < ndevices; i++) {
    config_t* config = configs + i;
    config->idevice = i;
    config->step = 0;
    config->nx = nx; config->ny = ny;
    config->inout_cpu = inout + np * i;
    config->status = thread_func(config);
  }
}
OpenMP. Linking

* gcc 4.3
* Command line
  * $ nvcc -Xcompiler \n    -fopenmp -Xlinker\n    -lgomp cudaOpenMP.cu
POSIX threads (pthreads)

- Implementation – library
- User is responsible for creating/completion of threads
- User manage threads communication explicitly
- Documents: https://computing.llnl.gov/tutorials/pthreads/man pthreads
POSIX threads (pthreads)

* Creating and waiting for threads
  `pthread_create, pthread_join`
* Critical section
  `pthread_mutex_lock, pthread_mutex_unlock, ...`
* Barriers and conditional waits
  `pthread_barrier_wait, pthread_cond_wait`
* static **CUT_THREADPROC** solverThread(SomeType *plan) {
    // Init GPU
    cutilSafeCall( cudaSetDevice(plan->device) );
    // start kernel
    SomeKernel<<<GS, BS>>>(some parameters);
    cudaMemcpyDeviceToHost(some data);

    cudaThreadExit();
    **CUT_THREADEND**;
}

Portability between Windows and Linux.
SomeType solverOpt[MAX_GPU_COUNT];
CUTThread threadID[MAX_GPU_COUNT];

for(i = 0; i < GPU_N; i++){
    solverOpt[i].device = i; ...
}

//Start CPU thread for each GPU
for(gpuIndex = 0; gpuIndex < GPU_N; gpuIndex++){
    threadID[gpuIndex] =

    cutStartThread((CUT_THREADROUTINE)solverThread, &SolverOpt[gpuIndex]);
}

//waiting for GPU results
 cutWaitForThreads(threadID, GPU_N);
MPI – Message Passing Interface

- Implementation – library, daemons
- MPI daemons control launching/state of MPI processes on cluster nodes
- The same program is launched on cluster nodes

- Launching program and creating threads
  **mpirun, mpiexec** (mpirun –np 4 ./MPI_calc_PI.exe)

- Initialization, deinitialization
  **MPI_Init, MPI_Finalize**

- Data transfer operations
  **MPI_Send, MPI_Recv, MPI_Bcast, ...**

- Synchronization
  **MPI_Barrier, ...**
Device address could be passed to MPI routines (CUDA 4.0+)

Available for OpenMPI trunk (Rolf vandeVaart)

```
[arom@cuda ~/dist]$
svn co http://svn.open-mpi.org/svn/ompi/trunk ompi-trunk
[arom@cuda ~/dist]$ cd ompi-trunk/
[arom@cuda ompi-trunk]$ ./autogen.pl
[arom@cuda ompi-trunk]$ mkdir build
[arom@cuda ompi-trunk]$ cd build
[arom@cuda build]$ ../configure --prefix=/home/dmikushin/opt/openmpi_gcc-trunk --with-cuda
[arom@cuda build]$ make install
```
Create CUDA context prior MPI_Init

http://www.open-mpi.org/faq/?category=running#mpi-cuda-support

cudaSetDevice(getenv("OMPI_COMM_WORLD_LOCAL_RANK")%cudaGetDeviceCount());

Version

* MVAPICH2
* OpenMPI
* Platform MPI
in MPI_Send/_Recv – device- pointers din1/din2

float *din1, *din2;
cuda_status = cudaMalloc((void**)&din1, size);
...
cuda_status = cudaMalloc((void**)&din2, size);
...
MPI_Request request;
int inext = (iprocess + 1) % nprocesses;
int iprev = iprocess - 1; iprev += (iprev < 0) ? nprocesses : 0;

Pass entire process input device buffer directly to input device buffer of next process.

mpi_status = MPI_Isend(din1, n*n, MPI_FLOAT, inext, 0, MPI_COMM_WORLD, &request);
mpi_status = MPI_Recv(din2, n*n, MPI_FLOAT, iprev, 0, MPI_COMM_WORLD, NULL);
mpi_status = MPI_Wait(&request, MPI_STATUS_IGNORE);
IPC – Inter-process communication

* Implementation - library
* User is responsible for processes creating/completion as well as process’ properties
  * `fork()`, `exit()`, …
* User manage threads communication explicitly
  * Shared memory, condition variables, signals, …
* Documentation:
  * man ipc
// Call fork to create another process.
// Standard: "Memory mappings created in the parent
// shall be retained in the child process."

pid_t fork_status = fork();

// From this point two processes are running the same code,
// if no errors.
if (fork_status == -1){
    fprintf(stderr, "Cannot fork process, errno = %d\n", errno);
    return errno;
}

// By fork return value we can determine the process role:
// master or child (worker)
int master = fork_status ? 1 : 0, worker = !master;

// Get the process ID
int pid = (int) getpid();
Creating context (\texttt{cuCtxCreate}). Created context becomes current. Device could have several contexts.

Context could be detached with \texttt{cuCtxPopCurrent} (‘floating’) and attached again (\texttt{cuCtxPushCurrent}).

When necessary context should to be destroyed (\texttt{cuCtxDestroy})

(!!!) creating context prior fork() results in undefined behavior.
for(int i=0; i<nGPUS; i++){
    CUdevice dev;
    CUresult cu_status = cuDeviceGet(&dev, i);
    if (cu_status != CUDA_SUCCESS) { /* Error handling */ }
    
    device_t *device = &devices[i];
    cu_status = cuCtxCreate(divice->ctx, 0, dev);
    if (cu_status != CUDA_SUCCESS) { /* Error handling */ }
    
    CUresult cu_status = cuCtxPopCurrent(divice->ctx);
    if (cu_status != CUDA_SUCCESS) { /* Error handling */ }
}
for(int i=0; i<nGPUS; i++){
    device_t *device = &devices[i];
    // set context active/current
    CUBresult cu_status = cuCtxPushCurrent(device->ctx);
    if (cu_status != CUDA_SUCCESS) {/* Error handling */ }

    // allocate memory, launch kernels

    // set context inactive
    cu_status = cuCtxPopCurrent(device->ctx);
    if (cu_status != CUDA_SUCCESS) {/* Error handling */ }
}

Working with context
for(int i=0; i<nGPUS; i++){
    device_t *device = &devices[i];
    // set context active/current
    CUresult cu_status = cuCtxPushCurrent(device->ctx);
    if (cu_status != CUDA_SUCCESS) {/* Error handling */}
    // wait for kernels to complete
    cuda_status = cudaThreadSynchronize();

    // save result, free memory...

    // Destroy context
    cu_status = cuCtxDestroy (device->ctx);
    if (cu_status != CUDA_SUCCESS) {/* Error handling */}
}
Conclusion

* It is possible to use multiple GPUs
* Multi-thread programming is required (CUDA 3.2)
  * Alternative is usage of driver functions.
* CPU thread can operate with several GPUs (CUDA 4.0)
* UVA allows one not to use Host memory for copying data between GPUs