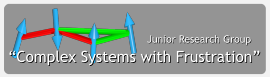
Computational Physics with GPUs

Lecture 5: Advanced techniques

Martin Weigel

Applied Mathematics Research Centre, Coventry University, Coventry, United Kingdom and
Institut für Physik, Johannes Gutenberg-Universität Mainz, Germany

41st Heidelberg Physics Graduate Days
Heidelberg, October 8–12, 2018



Outline

Slides and exercises

Check out the lecture notes and example code at

`http://users.complexity-coventry.org/~weigel/GPU/`

Any questions? Contact me at

`Martin.Weigel@mail.com`

OpenACC

OpenACC is an alternative scheme to CUDA (and OpenCL) with the following features:

- somewhat lighter in touch, less code is required
- available for Fortran, C and C++
- based on preprocessor directives: `#pragma acc`
- in its current (Nvidia) implementation, it relies on the 'PGI Community Edition',

`https://developer.nvidia.com/openacc-toolkit`

It is often well suited for accelerating existing applications with GPUs or for programmers that don't want to be too much bothered by implementation details.

OpenACC example

A simple code for matrix multiplication could be accelerated using OpenACC as follows:

```
void computeAcc(float *p, const float* M, const float *N,
                int Mh, int Mw, int Nw)
{
  #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Nw*Mw])
                            copyout(P[0:Mh*Nw])
  for (int i = 0; i < Mh; ++i) {
    #pragma acc loop
    for(int j = 0; j < Nw; ++j) {
      float sum = 0;
      for (int k = 0; k < Mw; ++k) {
        sum += M[i*Mw + k] * N[k*Nw+j];
      }
      P[i*Nw+j] = sum;
    }
  }
}
```

This is almost identical to the CPU code and, in fact, using a compiler that ignores `#pragma acc` it is perfectly valid C code!

OpenACC: advantages

The OpenACC programming model has a number of advantages:

- it is often possible to first write a sequential version of the program and then accelerate it using OpenAcc ⇒ much lower entry barrier for GPU computing
- it is much more straightforward to accelerate existing, legacy C, C++ or Fortran codes without rewriting them completely
- the fact that ignoring `#pragma acc` when using a standard compiler means that there is always an equivalent sequential program, which makes the code easier to debug — but it also means that some forms of parallelism will not be used in such codes

OpenACC: Execution and memory models

The execution model has three layers:

- execution units (different GPUs)
- threads
- vector operations for a single thread

The only synchronization supported is through fork and join.

Parallel execution is offloaded to GPU using the `parallel` or `kernels` construct.

The memory model assumes that host and device memory are completely distinct and independent. All data needs to be copied in and out explicitly. Data is accessible by all threads on a given execution unit (corresponding to global memory in CUDA). Faster memories such as caches and shared memory are only used by code automatically generated by the compiler.

OpenACC: Basic elements

The basic elements of OpenACC are as follows:

`#pragma acc parallel`
The following block is moved into the accelerator and executed using gangs of workers. Gangs are used for the outer level of parallelism and workers form a (possible) inner level. The corresponding numbers can be specified via `num_gangs` and `num_workers`. If these are not specified, these will be picked up at runtime.

`#pragma acc loop`
A following loop can be executed in parallel using the gangs or workers previously created.

```
#pragma acc parallel num_gangs(1024)
{
  #pragma acc loop gang
  for (int i = 0; i < 2048; ++i) {
    ...
  }
}
```

In this case, each gang lead will perform two iterations.

A second level of loop, if parallelized using `#pragma acc loop` will be assigned to workers instead of gangs if nothing different is specified explicitly.

OpenACC: Basic elements (cont'd)

To activate the third level of parallelism (vector or SIMD instructions), one can specify a loop as `#pragma acc parallel vector`.

On a CUDA device, gangs could be mapped to blocks, workers to warps and vector elements to threads within warps, but this is implementation dependent.

`#pragma acc kernels`
This is somewhat more general than `#pragma acc parallel` in that the following region can contain several kernels instead of just one. Each kernel can use a different number of gangs and workers. Also, loops in such regions without preceding `#pragma acc loop` will only be executed by one thread.

`restrict` and `independent`
It is not always possible for a compiler to decide whether a loop can be parallelized. Using the `independent` qualifier, we can enforce this.

```
#pragma acc loop independent
for (int l =0; l < m; ++l) {
  x[l] = x[l+n] + 1;
}
```


OpenACC: Basic elements (cont'd)

Data is explicitly moved in or out using the `copyin` and `copyout` constructs shown above.

`create`
The qualifier `create` is used to declare that variables do not need to be copied in or out but just need created (allocated) on the device (for example for temporary storage required during the calculation).

`deviceptr`
If storage on device is created by other means (for example through combining CUDA and OpenACC in one program), such arrays can be declared to OpenACC using the `deviceptr` statement.

`data`
This construct can be used to advise OpenACC of keeping the corresponding variables active in between successive calls to a given kernel.

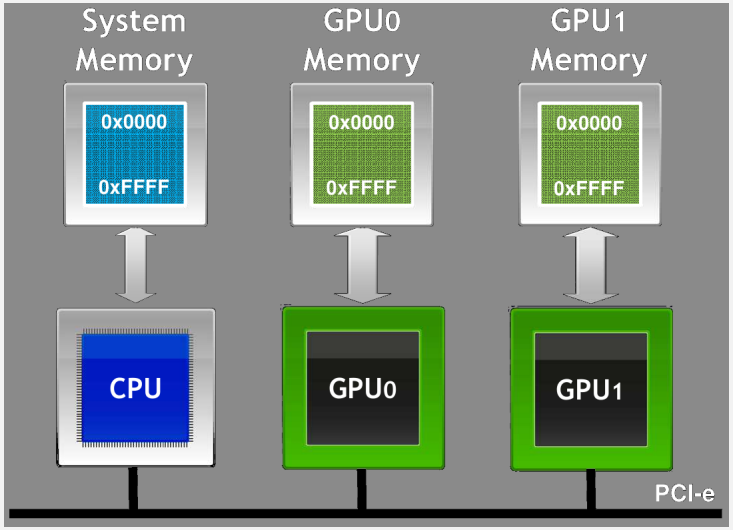
`async`
It is possible to make the accelerated execution of a region asynchronous, such that the control flow of the main program continues. Later synchronization points can be used to wait for the parallel part to have finished.

OpenACC: disadvantages

The OpenACC programming model also has a number of disadvantages:

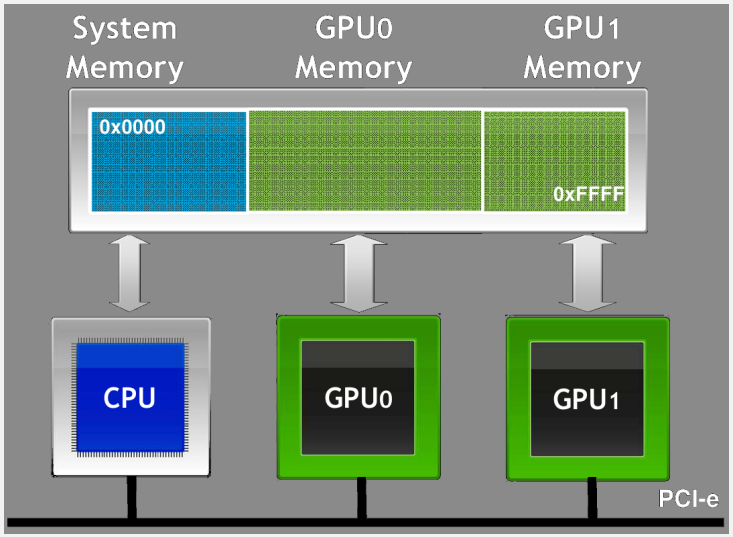
- a much less fine-grained control over what is happening
- no support for unified virtual memory and similar concepts
- in most cases it is not possible to get the maximum performance out of a device

Unified virtual addressing



Available only on some platforms!

Unified virtual addressing



Available only on some platforms!

Unified virtual addressing (cont'd)

CUDA automatically knows in which address space pointers belong, so can use

```
float *A_h, *A_d;
...
cudaMemcpy(A_h, A_d, cudaMemcpyDefault);
cudaMemcpy(A_d, A_h, cudaMemcpyDefault);
```

(Can find out from `cudaDeviceProp::unifiedAddressing` flag.)

It is also possible to copy directly between different GPUS (on the same node), see GPU direct below,

```
cudaMemcpy(gpu0_buf, gpu1_buf, buf_size, cudaMemcpyDefault);
```


Unified virtual addressing (cont'd)

Ultimately, it is also possible to write code that does not explicitly copy data between host and device.

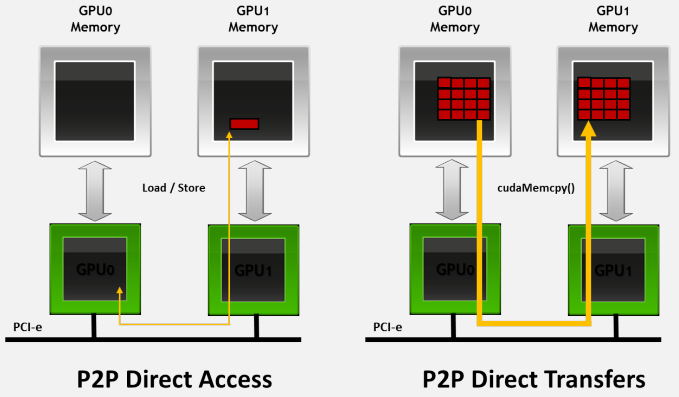
Compare the following traditional code

```
__global__ void AplusB( int *ret, int a, int b)
{ ret[threadIdx.x] = a + b + threadIdx.x; }
int main() {
    int *ret;
    cudaMalloc(&ret, 1000 * sizeof(int));
    AplusB<<< 1, 1000 >>>(ret, 10, 100);
    int *host_ret = (int *)malloc(1000 * sizeof(int));
    cudaMemcpy(host_ret, ret, 1000 * sizeof(int), cudaMemcpyDefault);
    for(int i=0; i<1000; i++) printf("%d: A+B = %d\n", i, host_ret[i]);
    free(host_ret);
    cudaFree(ret);
}
```

To a variant without explicit copies,

```
int main() {
    int *ret;
    cudaMallocManaged(&ret, 1000 * sizeof(int));
    AplusB<<< 1, 1000 >>>(ret, 10, 100); cudaDeviceSynchronize();
    for(int i=0; i<1000; i++) printf("%d: A+B = %d\n", i, ret[i]);
    cudaFree(ret);
}
```


GPUDirect



- P2P transfers: direct copies from GPU to GPU inside a node
- direct communication with network adaptors (using a pinned buffer in host memory shared between GPU and network adaptor)
- RDMA: direct access to remote GPU memories (completely avoiding the use of host memory)
- Limited to Tesla GPUs

GPUDirect: Peer-to-peer transfers

```
//Check for peer access between participating GPUs:
cudaDeviceCanAccessPeer(&can_access_peer_0_1, gpuid_0, gpuid_1);
cudaDeviceCanAccessPeer(&can_access_peer_1_0, gpuid_1, gpuid_0);

//Enable peer access between participating GPUs:
cudaSetDevice(gpuid_0);
cudaDeviceEnablePeerAccess(gpuid_1, 0);
cudaSetDevice(gpuid_1);
cudaDeviceEnablePeerAccess(gpuid_0, 0);

//UVA memory copy:
cudaMemcpy(gpu0_buf, gpu1_buf, buf_size, cudaMemcpyDefault);
```

Direct DMA copy avoiding host memory. Falls back to regular copy if P2P not available.

GPUDirect: direct P2P memory access

It is even possible to directly access memories residing on different GPUs.

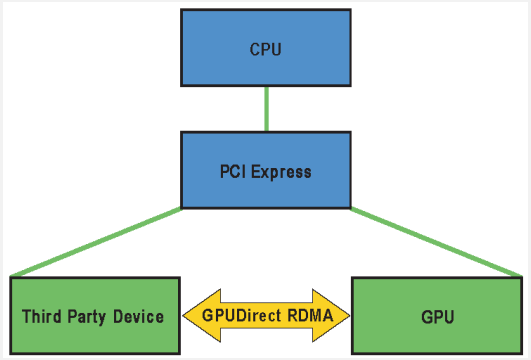
```
__global__ void SimpleKernel(float *src, float *dst)
{
  const int idx = blockIdx.x * blockDim.x + threadIdx.x;
  dst[idx] = src[idx];
}

cudaSetDevice(gpuid_0);
SimpleKernel<<<blocks, threads>>> (gpu0_buf, gpu1_buf);
```

(One needs to have enabled peer access as before.)

GPUDirect: RDMA

DMA is a well-known technique for devices to write directly to the CPU memory. RDMA enables this for GPUs



If a system is correspondingly configured, this allows for quite efficient integration with MPI.

The message passing interface (MPI)

The standard library for parallel programming on cluster machines (no shared memory); established in 1993.

MPI programs are launched with a configurable number of processes, typically running on different machines. All communication needs to be explicitly initiated. This can be

- one-to-one, one-to-many (scatter), many-to-one (gather)
- synchronous, asynchronous, or buffered
- involve barrier operations (process synchronization)

It is beyond the scope if these lectures to introduce MPI properly.

An MPI application can be run from the commandline via `mpirun`,

```
mpirun -np 16 ./myapplication <arguments>
```

MPI example

```
int main(int argc, char *argv[])
{
    char message[20];
    int myrank, tag=99;
    MPI_Status status;

    /* Initialize the MPI library */
    MPI_Init(&argc, &argv);
    /* Determine unique id of the calling process of all processes
        participating
       in this MPI program. This id is usually called MPI rank. */
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    if (myrank == 0) {
        strcpy(message, "Hello, there");
        /* Send the message "Hello, there" from rank 0 to rank 1. */
        MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, tag,MPI_COMM_WORLD)
            ;
    } else {
        /* Receive a message of at most 20 characters from process 0. */
        MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
        printf("received %s\n", message);
    }
    /* Finalize the MPI library to free resources acquired by it. */
    MPI_Finalize();
    return 0;
}
```


CUDA aware MPI

When running codes using GPUs on a cluster, one often needs to send the content of GPU buffers between MPI nodes.

With a correspondingly set up MPI installation, it is possible to include pointers to GPU memory in MPI function calls. Without CUDA-aware MPI, one would need to proceed as follows:

```
//MPI rank 0
cudaMemcpy(s_buf_h,s_buf_d,size,cudaMemcpyDeviceToHost);
MPI_Send(s_buf_h,size,MPI_CHAR,1,100,MPI_COMM_WORLD);

//MPI rank 1
MPI_Recv(r_buf_h,size,MPI_CHAR,0,100,MPI_COMM_WORLD, &status);
cudaMemcpy(r_buf_d,r_buf_h,size,cudaMemcpyHostToDevice);
```

While with CUDA-enabled MPI, the copies are not required,

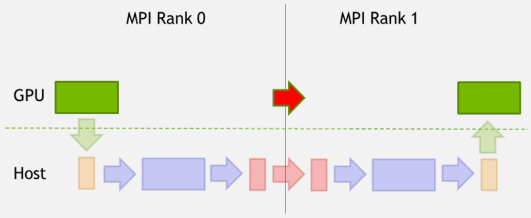
```
//MPI rank 0
MPI_Send(s_buf_d,size,MPI_CHAR,1,100,MPI_COMM_WORLD);

//MPI rank n-1
MPI_Recv(r_buf_d,size,MPI_CHAR,0,100,MPI_COMM_WORLD, &status);
```


CUDA aware MPI (cont'd)

This is based on a combination of the previously discussed techniques:

- unified virtual addressing
- GPUDirect using a shared buffer with the network adaptor
- GPUDirect using RDMS to send directly from GPU memory to the network
- GPUDirect P2P communications for MPI ranks running on the same node



Programming guidelines

What should I definitely keep in mind when trying to write a (good) GPU code?

- **Memory coalescence**: threads in a warp should prefer to access memory in the same cache line, ideally successive locations.
- **Parallel slack**: set up your kernels to use at least of the order of 10 times the number of available physical cores to facilitate latency hiding.
- **Occupancy**: choose your execution configuration and kernel set up to come reasonably close to each SM being loaded with the maximum number of allowed concurrent threads. Possibly use the CUDA occupancy calculator.
- **Shared memory**: if each memory location is read or written several times, it might be useful to pre-load tiles in shared memory. Or use shared memory for intermediate calculations that are done cooperatively by threads of a block.
- **Arithmetic density and data compression**: aim for a sufficient arithmetic density of the code. Sometimes it is cheaper to re-calculate something than to store it in global memory. Data compression helps to reduce memory bandwidth pressure.
- **Floating-point calulcations**: which accuracy is required? Are single-precision calculations sufficient? Check whether the fast intrinsics can be used.
- **Thread divergence**: avoid if possible.

Summary and outlook

This lecture

We have now taken a look at some more advanced features of GPU computing, interesting for the integration of CPU and GPU codes on big machines and for making legacy serial code make use of available GPUs.

Everything

You should now be ready to go for your own small and big GPU projects. Please keep me posted!

Reading

Additional to the other sources, for some general comments regarding massively parallel computing see also the recent review

M. Weigel, Monte Carlo methods for massively parallel computers, arXiv:1709.04394.

Exercise

Start playing with `mpi` and its integration with CUDA.

- Familiarize yourself with how to compile and run a simple MPI program on the machine of interest.
- Check how to write a CUDA aware MPI program. Either compile the MPI parts and CUDA parts separately with `mpicc` and `nvcc` and then link or compile everything with `nvcc`.
- Write an MPI/CUDA version of the vector addition code (or one of the other examples).

```
nvcc -I/usr/include/mpi -lmpi mpi.cu -o mpi
mpirun -np 2 ./mpi
```