Best Practice mini-guide
accelerated clusters
Using General Purpose GPUs
Alan Gray, EPCC
Anders Sjöström, LUNARC
Nevena Ilieva-Litova, NCSA
Partial content by CINECA: http://www.hpc.cineca.it/content/gpgpu-general-purpose-graphics-processing-unit
Editor: Maciej Szpindler
ICM University of Warsaw
May 2013
# Table of Contents

1. Introduction ........................................................................................................... 3
2. Architecture and Configurations ........................................................................... 3
   2.1. Overview of general architecture of GPU clusters ........................................... 3
   2.2. Specific information on Tier-1 accelerated clusters ........................................... 3
      2.2.1. Eurora and PLX accelerated clusters at CINECA .................................. 3
      2.2.2. MinoTauro accelerated clusters at BSC .................................................. 3
      2.2.3. Laki accelerated cluster and Hermit Supercomputer at HLRS ................... 4
      2.2.4. Cane accelerated cluster at PSNC .......................................................... 4
      2.2.5. Accessing GPU accelerated system with PRACE RI ............................... 4
3. GPU Programming with CUDA .............................................................................. 4
   3.1. Offloading Computation to the GPU ............................................................... 4
   3.2. Memory Management ...................................................................................... 7
   3.3. Synchronization .............................................................................................. 8
4. Best practice for optimizing codes on GPUs .......................................................... 8
   4.1. Minimizing Data Transfer Overhead .............................................................. 8
   4.2. Occupancy and Memory Latency Hiding ...................................................... 9
   4.3. Minimizing Memory Latency ......................................................................... 9
   4.4. Maximizing Memory Bandwidth ................................................................... 10
   4.5. Use of on-chip Memory ............................................................................... 10
5. Hybrid programming with GPUs and multi-GPUs .................................................. 11
   5.1. Hardware generation ...................................................................................... 11
   5.2. CUDA features .............................................................................................. 11
   5.3. Hyper-Q ......................................................................................................... 11
   5.4. Dynamic parallelism ...................................................................................... 11
   5.5. RDMA .......................................................................................................... 12
   5.6. Virtual addressing ........................................................................................ 13
   5.7. CUDA aware MPI ........................................................................................ 14
   5.8. Multi CPU/GPU .......................................................................................... 15
   5.9. Debugging and Profiling .............................................................................. 15
   5.10. Example multi-core Multi-GPU ................................................................... 15
6. GPU libraries ........................................................................................................ 15
   6.1. The CUDA Toolkit 5 .................................................................................... 15
      6.1.1. CUDA Runtime and Math libraries ......................................................... 16
      6.1.2. CuFFT .................................................................................................... 16
      6.1.3. CuBLAS ................................................................................................. 16
      6.1.4. CuSPARSE ............................................................................................ 16
      6.1.5. CuRAND .............................................................................................. 17
      6.1.6. NPP ....................................................................................................... 17
      6.1.7. Thrust .................................................................................................... 17
   6.2. Other libraries ............................................................................................... 17
      6.2.1. CULA .................................................................................................... 17
      6.2.2. NVIDIA Codec libraries ....................................................................... 17
      6.2.3. CUSP .................................................................................................... 17
      6.2.4. MAGMA ............................................................................................... 18
      6.2.5. ArrayFire ............................................................................................. 18
7. Other programming models for GPUs .................................................................. 18
   7.1. Programming GPUs using Directives .......................................................... 18
1. Introduction

A GPU is a specialized device designed to rapidly manipulate high amounts of graphical pixels. Historically, GPU were born for being used in advanced graphics and videogames.

More recently interfaces have been built to interact with codes not related to graphical purposes, for example for linear algebraic manipulations.

*General-purpose GPU computing or GPGPU* computing is the use of a GPU (graphics processing unit) to do general purpose scientific and engineering computing.

The model for GPU computing is to use a CPU and GPU together in a heterogeneous co-processing computing model. The sequential part of the application runs on the CPU and the computationally-intensive part is accelerated by the GPU. From the user’s perspective, the application just runs faster because it is using the high-performance of the GPU to boost performance.

The GPU has evolved over the years to have teraflops of floating point performance.

The success of GPGPUs in the past few years has been the ease of programming of the associated CUDA parallel programming model. In this programming model, the application developer modify their application to take the compute-intensive kernels and map them to the GPU. The rest of the application remains on the CPU. Mapping a function to the GPU involves rewriting the function to expose the parallelism in the function and adding “C” keywords to move data to and from the GPU.

2. Architecture and Configurations

2.1. Overview of general architecture of GPU clusters

The *Tesla 20-series* NVIDIA GPU is based on the “Fermi” architecture. Fermi is optimized for scientific applications with key features such as 500+ gigaflips of IEEE standard double precision floating point hardware support, L1 and L2 caches and ECC memory error protection.

The *Tesla K-series* NVIDIA GPU is based on the “Kepler” architecture, which is the latest CUDA architecture with improved performance and power efficiency.

2.2. Specific information on Tier-1 accelerated clusters

2.2.1. Eurora and PLX accelerated clusters at CINECA

For complete information on GPU accelerated resources at CINECA refer to: [http://www.hpc.cineca.it/content/gpgpu-general-purpose-graphics-processing-unit#gpgpuatCineca](http://www.hpc.cineca.it/content/gpgpu-general-purpose-graphics-processing-unit#gpgpuatCineca).

The GPU resources of the Eurora cluster consist of 2 nVIDIA Tesla K20 "Kepler" per node, with compute capability 3.x.

The GPU resources of the cluster PLX consist of 2 NVIDIA Tesla M2070 GPUs "Fermi" per node with compute capability 2.0. Ten of the nodes are equipped with 2 NVIDIA QuadroPlex 2200 S4. The GPUs are configured with the Error Correction Code (ECC) support active, that offers protection of data in memory to enhance data integrity and reliability for applications. Registers, L1/L2 caches, shared memory, and DRAM all are ECC protected.

All tools and libraries required in the GPU programming environment are contained in the CUDA tooling. The CUDA toolkit is made available through the “cuda” module. When need to start a programming environment session with GPUs, the first thing to do is to load the CUDA module.

2.2.2. MinoTauro accelerated clusters at BSC

For complete information on GPU accelerated resources at BSC refer to: [http://www.bsc.es/marenostrum-support-services/other-hpc-facilities/nvidia-gpu-cluster](http://www.bsc.es/marenostrum-support-services/other-hpc-facilities/nvidia-gpu-cluster)
MinoTauro is a NVIDIA cluster with 128 Bull B505 blades equipped with 2 M2090 NVIDIA GPU Cards.

### 2.2.3. Laki accelerated cluster and Hermit Supercomputer at HLRS

For complete information on GPU accelerated resources at HLRS refer to: http://www.hlrs.de/systems/platforms/.

Laki is a Nehalem based cluster with 32 Nvidia Tesla S1070 GPU nodes.

Hermit is a Cray XE6 Tier-0 Supercomputer with GPU accelerated nodes.

### 2.2.4. Cane accelerated cluster at PSNC

For complete information on GPU accelerated resources at PSNC refer to: https://hpc.man.poznan.pl/modules/resourcesection/item.php?itemid=61.

Cane is Opteron accelerated cluster with 334 NVIDIA TeslaM2050 GPU modules.

### 2.2.5. Accessing GPU accelerated system with PRACE RI

For more recent information on GPGPU systems available follow the PRACE DECI Calls. More information is available on the PRACE website: http://prace-ri.eu

### 3. GPU Programming with CUDA

The CUDA programming model defines language extensions which allow NVIDIA GPUs to be programmed in C and C++. Included is functionality for defining and launching kernels to execute using multiple threads concurrently on the GPU, plus the necessary facilities for transferring data between the distinct CPU and GPU memory spaces.

#### 3.1. Offloading Computation to the GPU

GPUs work under the "Stream" model of computation:

**Figure 1. "Stream" model**

The data set is decomposed into a stream of elements, and a single computational function (or kernel) operates on each element. A thread is defined as execution of kernel on one data element. Multiple cores can process multiple elements in parallel (i.e. many threads can run in parallel). This model is only suitable for data-parallel problems.
NVIDIA GPUs architecturally have a 2-level hierarchy:

**Figure 2. GPU architecture hierarchy**

![Diagram of GPU architecture hierarchy]

Each chip comprises multiple Stream Multiprocessors (SMs), each featuring multiple cores. In CUDA, this is abstracted as Grid of Thread Blocks, where the multiple blocks in a grid map onto the multiple SMs, and each block contains multiple threads, mapping onto the cores in an SM. When programming in CUDA it is not necessary to know the exact details of the hardware (i.e. number of SMs or cores per SM), since typically it is best to oversubscribe (i.e. use more blocks than SMs, and more threads than cores), and the system will perform scheduling automatically. This allows the same code to be portable and efficient across different GPU versions (which may have different specific configurations).

As an example consider the following loop, in which two vectors are added:

```c
for (i=0;i<N;i++)
  c[i] = a[i] + b[i];
```

This is parallel in nature: we can decompose this for computation on the GPU by using a separate thread to perform each iteration. First, we replace this loop by a function:

```c
__global__ void vectorAdd(float *a, float *b, float *c){
  int i = threadIdx.x;
  c[i] = a[i] + b[i];
}
```

The `__global__` specifier is used to specify that this function is to form a GPU kernel. The internal CUDA variable `threadIdx.x` is unique to each thread in a block: this replaces the loop index.

Now we launch this kernel by calling the function on multiple CUDA threads using special CUDA bracketing syntax:
int blocksPerGrid=1; //use only one block
int threadsPerBlock=N; //use N threads in the block
vectorAdd<<blocksPerGrid, threadsPerBlock>>>(a, b, c);

The previous example only uses 1 block, i.e. only 1 SM on the GPU, so will be suboptimal. In practice, we need to use multiple blocks, e.g.:

__global__ void vectorAdd(float *a, float *b, float *c){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    c[i] = a[i] + b[i];
}
...
int blocksPerGrid=N/256; //assuming N divides 256 exactly
int threadsPerBlock=256;
vectorAdd<<blocksPerGrid, threadsPerBlock>>>(a, b, c);
...

The loop index is now replaced with a global thread index, calculated through use of threadIdx.x together with blockIdx.x, unique to each block in a grid, and blockDim.x, the number of threads per block. We have chosen to use 256 threads per block, which is typically a good number, but the optimal number for a specific code should be found through experimentation.

You can think of the use of multiple blocks as restructuring the original loop

for (i=0;i<N;i++){
    c[i] = a[i] + b[i];
}

as

for (i0=0;i0<(N/256);i0++){
    for (i1=0;i1<256;i1++){
        i = i0*256 + i1;
        c[i] = a[i] + b[i];
    }
}

and parallelising the inner loop over threads in a block, and the outer loop over blocks.

The previous examples were one dimensional. Each thread block can be 1D, 2D or 3D to best fit the algorithm, e.g. for matrix addition:

__global__ void matrixAdd(float a[N][N], float b[N][N],
float c[N][N]){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
\[ c[i][j] = a[i][j] + b[i][j]; \]

...  
\[
\text{dim3 blocksPerGrid}(N/16,N/16); // (N/16)x(N/16) blocks/grid (2D)  
\text{dim3 threadsPerBlock}(16, 16); // 16x16 threads/block (2D)  
\text{matrixAdd}<<<\text{blocksPerGrid, threadsPerBlock}>>>(a, b, c);  
\]

### 3.2. Memory Management

The GPU has a separate memory space from the host CPU:

**Figure 3. CPU/GPU memory scheme**

![CPU/GPU Memory Scheme](image)

We cannot simply pass normal C pointers to CUDA threads, but need to explicitly manage GPU memory and copy data to/from it before/after the kernel is launched. Analogous to the C `malloc` and `free` calls, `cudaMalloc` is used to allocate GPU memory and `cudaFree` releases it again, e.g.

```c
float *array_device;  
cudaMalloc(&array_device, N*sizeof(float));  
...  
cudaFree(a);  
```

Once memory has been allocated on the GPU, we need to be able to copy data to/from it. `cudaMemcpy`, analogous to the C `memcpy`, does this:

```c
\text{cudaMemcpy}(array_device, array_host, N*sizeof(float),  
\text{cudaMemcpyHostToDevice});  
\text{cudaMemcpy}(array_host, array_device, N*sizeof(float),  
\text{cudaMemcpyDeviceToHost});  
```
The first and second arguments specify the destination and source memory locations respectively. The third argument specifies the amount of data, and the final argument specifies the direction of transfer.

Transfers between host and device memory are relatively slow and can become a bottleneck, so should be minimized when possible.

### 3.3. Synchronization

Kernel calls are non-blocking: the host program continues immediately after it calls the kernel. This allows the overlap of computation on CPU and GPU. The `cudaThreadSynchronize()` call waits for the kernel to finish. Standard `cudaMemcpy` calls are blocking (but non-blocking variants exist).

Within a kernel, to synchronize between threads in the same block use the `syncthreads()` call. This allows, for example, threads in the same block to communicate through memory spaces that they share. For example, assuming $x$ is local to each thread and `array` is in a shared memory space, the value of $x$ can be communicated from thread 0 to thread 1 as follows:

```c
if (threadIdx.x == 0)
    array[0]=x;
    syncthreads();
if (threadIdx.x == 1)
    x=array[0];
```

Note that it is not possible to communicate between threads in different blocks in a kernel: you must instead exit the kernel and launch a new one.

### 4. Best practice for optimizing codes on GPUs

In order to get good performance on the GPU architecture, it is often necessary to perform code optimization. This section highlights some key GPU performance issues, and provides optimization advise. Concentration is on the NVIDIA GPU, but many concepts will be transferable to other accelerators.

#### 4.1. Minimizing Data Transfer Overhead

The CPU (host) and GPU (device) have separate memories. All data read/written on the device must be copied to/from the device (over the PCIe bus). This is very expensive, so it is important to try to minimize copies wherever possible, and keep data resident on device. This may involve porting more routines to device, even if they are not computationally expensive. For example, code such as the following

```c
Loop over timesteps
  inexpensive_routine_on_host(data_on_host)
  copy data from host to device
  expensive_routine_on_device(data_on_device)
  copy data from device to host
End loop over timesteps
```

would be optimized by porting the inexpensive routine to device and moving the data copies outside of the loop:
copy data from host to device
Loop over timesteps
  inexpensive_routine_on_device(data_on_device)
  expensive_routine_on_device(data_on_device)
End loop over timesteps
copy data from device to host

Also, in some cases it may be quicker to calculate quantities from scratch on the device instead of copying from the host.

4.2. Occupancy and Memory Latency Hiding

On NVIDIA GPUs, there are less instruction scheduling units than cores. Threads are scheduled in groups of 32, called a warp. Threads within a warp must execute the same instruction in lock-step (on different data elements). The CUDA programming allows branching, but this results in all cores following all branches, with only the required results saved. This is obviously suboptimal; it is important to avoid intra-warp branching wherever possible (especially in key computational sections). For example, suppose you want to split your threads into 2 groups, each to perform a separate task. The code

```c
i = blockIdx.x*blockDim.x + threadIdx.x;
if (i%2 == 0)
  ...  
else
  ...
```

would lead to threads within a warp diverging, whilst the alternative

```c
i = blockIdx.x*blockDim.x + threadIdx.x;
if ((i/32)%2 == 0)
  ... 
else
  ...
```

would result in threads within warp following same path, and would hence be more optimal.

4.3. Minimizing Memory Latency

When programming for the GPU architecture, the programmer decomposes loops to threads. Obviously, there must be at least as many total threads as cores, otherwise cores will be left idle. For best performance, we actually want the number of threads to be much greater than the number of cores. Accesses to global memory have several hundred cycles latency: when a thread stalls waiting for data, if another thread can switch in this latency can be hidden. NVIDIA GPUs feature very fast thread switching, and support many concurrent threads. The Fermi architecture supports up to 1536 concurrent threads on an SM, e.g. if there are 256 threads per block, it can run up to 6 blocks concurrently per SM (and the remaining blocks will queue). But note that resources must be shared between threads: high use of on-chip memory and registers will limit number of concurrent threads. Typically it is best to use many thousands of threads in total. The optimal number of threads per block should be found by experimentation. For example, the (serial) code

```
Loop over i from 1 to 512
```
Loop over \( j \) from 1 to 512
independent iteration

could be ported to the GPU using a 1D decomposition:

\[
\text{Calc } i \text{ from thread/block ID} \\
\text{Loop over } j \text{ from 1 to 512} \\
\text{independent iteration}
\]

or a 2D decomposition:

\[
\text{Calc } i \& j \text{ from thread/block ID} \\
independent \text{ iteration}
\]

The latter will be more optimal since it results in 262,144 threads, as opposed to just 512.

### 4.4. Maximizing Memory Bandwidth

The memory bandwidth for the graphics memory on the GPU is high compared to the CPU, but there are many data-hungry cores so memory bandwidth is still a performance bottleneck. The maximum bandwidth is achieved when data is loaded for multiple threads in a single transaction: this is called memory coalescing. This will happen when data access patterns meet certain conditions: 16 consecutive threads (i.e. a half-warp) must access data from within the same memory segment. This condition is met when consecutive threads read consecutive memory addresses within a warp. If the condition is not met, memory accesses are serialized, significantly degrading performance. Adapting code to allow coalescing can dramatically improve performance. For example, the code

```c
row = blockIdx.x*blockDim.x + threadIdx.x; 
for (col=0; col<N; col++) 
    output[row][col]=2*input[row][col];
```

will not result in coalesced memory accesses since consecutive threads belong to consecutive rows, but the data structures are stored in row-major format (in C), so consecutive row values do not have consecutive memory addresses. The alternative

```c
col = blockIdx.x*blockDim.x + threadIdx.x; 
for (row=0; row<N; row++) 
    output[row][col]=2*input[row][col];
```

will result in coalesced memory accesses, so will be more optimal.

### 4.5. Use of on-chip Memory

Expensive off-chip memory accesses can be avoided altogether if the on-chip memory and registers can be utilized, but these are relatively small resources. Each NVIDIA SM has a "Shared memory" space accessible to all threads in a block. The system will automatically use this as a cache, but it may be beneficial to perform manual caching through declaring variables with the __shared__ qualifier, and performing copies within the kernel. There also
exists read-only "Constant memory" on the GPU, which can be used through use of the `__constant__` qualifier and `cudaMemcpyToSymbol` library call (see the NVIDIA CUDA programming guide).

5. Hybrid programming with GPUs and multi-GPUs

This section concern hybrid programming and the use of multiple GPUs. As with the case of a single GPU code, care has to be taken in order to achieve good performance over multiple accelerators. In order to scale beyond what is possible on a single GPU some framework has to be utilized. possible frameworks are for instance, MPI or OpenMP. In this section we will look at possible solutions for the use of multiple GPUs.

5.1. Hardware generation

Depending on which generation of CUDA enabled cards are being used, different options are available to the programmer. In this section only the Kepler architecture is to be considered. The two main differentiating features between the Fermi and the Kepler architectures are the existence of "dynamic parallelism" and "Hyper Q" on the Kepler generation cards.

5.2. CUDA features

At the time of writing the latest version of the CUDA-library is 5.0. In order to give recommendations on the use of multiple GPUs a basic knowledge on the possibilities within the CUDA-framework is required.

- Hyper-Q: multiple threads or processes can launch work on a single GPU.
- Dynamic parallelism: Allowing a CUDA kernel to create and synchronize new nested work.
- GPUDirect: Allowing communication between devices.
- Unified Virtual Addressing: Unified memory address space.

5.3. Hyper-Q

Hyper-Q enable multiple threads or processes on the CPU to launch work on a single GPU simultaneously. This lead to better utilization of the GPU resources. The total number of connections between the host and the GPU is 32. Hyper-Q connections from multiple CUDA streams or from multiple MPI processes.

Figure 4. Hyper-Q multiple threads launching work on the GPU

5.4. Dynamic parallelism

Dynamic parallelism gives the ability for a CUDA kernel to create and synchronize new nested work, launch other kernels using the CUDA runtime API, synchronize on kernel completion, do device memory management, and create and use streams and events.
5.5. RDMA

RDMA introduced in CUDA 5.0 enables a direct path for communication between the GPU and a peer device via PCI Express. GPUDirect and RDMA enable direct memory access (DMA) between GPUs and other PCIe devices, Peer-To-Peer transfers between GPUs, and Peer-To-Peer memory access.

Figure 6. RDMA - direct communication between PCIe-devices
5.6. Virtual addressing

Unified virtual addressing (UVA) introduced in CUDA 4.0 is available on Fermi and Kepler class GPUs. Memory address management system is enabled by default in CUDA 4.0 and later releases on Fermi and Kepler GPUs. The virtual address (VA) range of the application is partitioned into two areas: the CUDA-managed VA range and the OS-managed VA range.
5.7. CUDA aware MPI

A quite recent possibility in multi GPU programming is to use CUDA aware MPI libraries. Several CUDA aware implementations are available:

- MVAPICH2 1.8/1.9b
- OpenMPI 1.7 (beta)
- CRAY MPI (MPT 5.6.2)
- IBM Platform MPI (8.3)

The idea is to avoid the coding overhead of copying the data from the device to the host but rather pass the GPU Buffers directly to MPI as shown below where the following code:

```c
//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);
```

is equivalent to:

```c
//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);

5.8. Multi CPU/GPU
Depending on the amount of effort the programmer is willing to put into a given project, different routes can be chosen. At the time of writing the least demanding approach on the programmer is to use OpenACC (as discussed in the other programming models section of this document) combined with either MPI or OpenMP. Using the combination of MPI or OpenMP together with CUDA is another option, as is using a CUDA aware MPI library.

5.9. Debugging and Profiling
As the complexity of debugging and profiling increases with the number of threads the absolute recommendation is to use a parallel debugger/profiler preferably with a graphical user interface to debug GPU accelerated code. Examples of parallel visual debuggers/profilers are Allinea DDT, NVIDIA Nsight (for Visual Studio or Eclipse) VampirTrace or TotalView.

5.10. Example multi-core Multi-GPU
To illustrate the effect of having an improper balance between the number of GPUs and CPUs/cores (in this case two cpus with 8 cores each) a commercial FEM-code has been run with a steady state problem with 500000 degrees of freedom. As can be seen in the table below the code scales fairly well using CPUs only. However, combining CPUs and GPUs the scaling is lost when too many cores are used in relation to the number of GPUs. In this case using more than one core per GPU doesn't give much performance. This is probably due to communication overhead as the sub-problem sizes are becoming too small. All times denoting wallclock time in seconds.

<table>
<thead>
<tr>
<th>Cores</th>
<th>0 GPU</th>
<th>1 GPU</th>
<th>2 GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>80064</td>
<td>9228</td>
<td>9433</td>
</tr>
<tr>
<td>2</td>
<td>40617</td>
<td>7665</td>
<td>6102</td>
</tr>
<tr>
<td>8</td>
<td>20056</td>
<td>7174</td>
<td>5427</td>
</tr>
<tr>
<td>16</td>
<td>12880</td>
<td>11561</td>
<td>8100</td>
</tr>
</tbody>
</table>

6. GPU libraries

6.1. The CUDA Toolkit 5
Within the CUDA programming model there are built-in libraries that provide efficient realization of different algorithms on the GPU. All libraries are gathered together in the CUDA Toolkit, the newest production release of which is the CUDA Toolkit 5. In the toolkit also a CUDA compiler and various tools for debugging and optimization are to be found (https://developer.nvidia.com/cuda-toolkit).

The CUDA Toolkit includes the following libraries:
- cuFFT - an implementation of the Fast Fourier Transform;
- cuBLAS - a complete BLAS library;
- cuSPARSE - a library for handling sparse matrices;
- cuRAND - a random number generator;
- NPP - a set of performance primitives for image and video processing;
• Thrust - template parallel algorithms and data structures and
• CUDA Runtime and Math Libraries that provide system calls and high performance mathematical routines.

The standard use of routines from the CUDA libraries follows this scheme: applications allocate the required variables in the GPU memory space, fill them with data, execute a sequence of desired functions on them and then return the results from the device memory to the memory of the host.

6.1.1. CUDA Runtime and Math libraries

CUDA Runtime Library contains important system calls needed for the operation of the device and other basic functions like, e.g., memory access. CUDA Mathematical Library is a collection of all C/C++ standard library mathematical functions that are supported in device code, as well as intrinsic functions (that are only supported in device code). This library is available by simply adding #include math.h in the source code. All functions are extensively tested, the information about the error bounds being also available, see: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#mathematical-functions-appendix.

6.1.2. CuFFT

CuFFT provides a simple interface for computing discrete Fourier transforms of complex and real-valued data sets on the GPU. The library provides an algorithm for every data input size and is highly optimized for input sizes that can be written in the form 2^a 3^b 5^c 7^d. There is a complex to complex, real to complex and complex to real input to output in 1D, 2D and 3D for transform sizes up to 128 million elements in single and 64 million elements in double precision in any dimension, limited only by the available device memory. For more details on cuFFT see: http://docs.nvidia.com/cuda/cufft/index.html.

6.1.3. CuBLAS

CuBLAS is the NVIDIA solution for a GPU-accelerated version of the complete standard BLAS (Basic Linear Algebra Subroutines) library, supporting all 152 standard BLAS routines using single, double, complex and double complex data types. It supports multiple GPU's and concurrent kernels. Since CUDA Toolkit version 5, the device API can be called also from CUDA kernels and there is a batched LU factorization API.

Within cuBLAS data is stored in column-major style and with 1-based indexing. For C/C++ applications macros should be defined to convert to their native array semantics in order to implement matrices on top of one-dimensional arrays. In case of natively written C/C++ code with 0-based indexing, the array index of a matrix element in row i and column j can be computed with the following macro:

```
#define Index2C(i,j,ld) (((j)*ld)+(i))
```

where ld is the leading dimension of the matrix, which in the case of column-major storage is the number of rows of the allocated matrix.

The cuBLAS library can be used via the legacy API when including the cublas.h header file. It is recommended to use the new version of the API for its greater functionality and for being more consistent with other CUDA libraries by including the cublas_v2.h header file. In addition, applications using the CuBLAS library need to link against the DSO cublas.so (Linux), the DLL cublas.dll (Windows), or the dynamic library cublas.dylib (Mac OS X). (http://docs.nvidia.com/cuda/cublas/index.html).

6.1.4. CuSPARSE

The NVIDIA CUDA Sparse Matrix library provides a set of basic linear algebra subroutines that operate with sparse matrices and is designed to be called from C or C++ applications. It supports several types of storage formats of the matrices – dense, coordinate, compressed sparse row, compressed sparse column and hybrid storage formats of float, double, complex and double complex data types. The library routines are classified in 3 levels of
operations, which can handle sparse vector by dense vector operations, sparse matrix by dense vector and sparse matrix by a set of dense vectors operations. There is also a conversion level, which includes operations that convert matrices in different matrix formats.

Other features of the library are the routines for sparse matrix by sparse matrix addition and multiplication, a sparse triangular solver and a tri-diagonal solver.

The cuSPARSE library can be used via the legacy API when including the cusparse.h header file. It is recommended to use the new version of the API for its greater functionality and for being more consistent with other CUDA libraries by including cusparse_v2.h header file. Applications using the CUBLAS library need to link against the DSO cusparse.so (Linux), the DLL cusparse.dll (Windows), or the dynamic library cusparse.dylib (Mac OS X) (http://docs.nvidia.com/cuda/cusparse/index.html).

6.1.5. CuRAND

CuRAND is the NVIDIA CUDA library for random number generation. It is very flexible and consists of two parts - a standard CPU library for execution on the host, which is accessible via the curand.h header file and device library for the GPU, accessible via the curand_kernel.h header file. Random numbers can be generated on the host or on the device. In this case, other user written kernels can use the random numbers without having to copy them in the global memory of the device or to transfer them to the host and back.

There are seven types of random number generators in cuRAND that fall in two categories – pseudo-random number and quasi-random number generators, including the MRG32k3a, MTGP, XORWOW and Sobol algorithms with adjustable options. Random number generators support sampling from uniform, normal, log-normal and Poisson distributions in single and double precision (http://docs.nvidia.com/cuda/curand/index.html).

6.1.6. NPP

NPP is the NVIDIA CUDA Performance Primitives library and contains over 1900 image processing, 600 signal processing and numerous video processing routines. Main feature of the library is that it is written in a way that eliminates unnecessary data transfer to/from the CPU memory. It also supports data exchange and initialization, arithmetic and logical operations, color conversions, several filter functions, jpeg functionality, geometry transforms, statistics functions and more (https://developer.nvidia.com/sites/default/files/akamai/cuda/files/CUDA-ownloads/NPP_Library.pdf).

6.1.7. Thrust

Thrust is a C++ template library for CUDA, based on the Standard Template Library (STL). It contains a collection of parallel algorithms and data structures, which provide a high-level interface for GPU programming (http://docs.nvidia.com/cuda/thrust/index.html).

6.2. Other libraries

There are also other libraries for accelerating CUDA applications delivered both from NVIDIA, as well as from third parties.

6.2.1. CULA

The CULA libraries contain linear algebra routines for both dense and sparse matrices.

6.2.2. NVIDIA Codec libraries

The NVIDIA Codec libraries provide tools for video encoding and decoding on NVIDIA GPU’s.

6.2.3. CUSP

The CUSP library is a C++ template library for sparse matrix computations, developed by Nathan Bell and Michael Garland.
6.2.4. MAGMA

MAGMA is a dense linear algebra library for heterogeneous/hybrid architectures such as "multicore+GPU" systems.

6.2.5. ArrayFire

The ArrayFire library contains functions for mathematical operations, signal and image processing, statistics and more and is available for C, C++ and Fortran programming languages. It can be used on AMD, Intel and NVIDIA hardware both under CUDA and OpenCL.

7. Other programming models for GPUs

7.1. Programming GPUs using Directives

Language extensions such as Cuda and OpenCL are very powerful since they give programmers the flexibility and control to interface with the GPU at a low level. However, the development process is often tricky and time consuming, and results in complex and non-portable code. An alternative approach is to allow the compiler to automatically accelerate code sections on the GPU (including decomposition, data transfer, etc). For real applications, compilers are typically not intelligent enough (or simply can’t deduce the required information from the source code) to generate optimal GPU-accelerated code, so there must be a mechanism for the programmer to provide the compiler with hints regarding which sections to be accelerated, available parallelism, data usage patterns etc. Directives provide this mechanism: these consist of a special syntax which is understood by accelerator compilers and ignored (treated as code comments) by non-accelerator compilers. This means in principle the same source code can be compiled for either a GPU-accelerated architecture or a traditional CPU-only machine.

Several accelerator compilers have emerged over the last few years including the PGI Accelerator Compiler, the CAPS HMPP compiler and the Cray compiler. These variants initially involved programming models which differ syntactically but are similar conceptually. The OpenACC standard was announced in November 2011 By CAPS, CRAY, NVIDIA and PGI: this is a standardised model which is supported by all of the above products. The remainder of this section illustrates accelerator directives using OpenACC. For a definitive guide including full list of available directives, clauses and options please see the documentation at www.openacc-standard.org.

With directives inserted, the compiler will attempt to compile the key kernels for execution on the GPU, and will manage the necessary data transfer automatically. The format of these directives is as follows:

C:

```c
#pragma acc ...
```

Fortran:

```fortran
!$acc ...
```

These are ignored by non-accelerator compilers.

The programmer specifies which regions of code should be offloaded to the accelerator with the `parallel` construct:

C:

```c
#pragma acc ...
```
This directive is usually not sufficient on its own to create efficient GPU-accelerated code: it needs (at least) to be combined with the `loop` directive.

The `loop` directive is applied immediately before a loop, specifying that it should be parallelised on the accelerator:

C:

```c
#pragma acc loop
for(…){
  …loop body…
}
```

Fortran:

```fortran
!$acc loop
do ...
  …loop body...
end do
```

The `loop` construct must be used inside a parallel construct:

C:

```c
#pragma acc parallel
{
  …
```
Multiple **loop** constructs may be used within a single **parallel** construct.

The **parallel loop** construct is shorthand for the combination of a **parallel** and (single) **loop** construct.

**C:**

```c
#pragma acc parallel loop
for (...) {
    ...loop body...
}
```

**Fortran:**

```fortran
!$acc parallel loop
do ...
    ...loop body...
end do
!$acc end parallel loop
```

Here is an example of a code being accelerated through use of the **parallel loop** construct:

```fortran
!$acc parallel loop
do i=1, N
    output(i)=2.*input(i)
end do
```
The compiler automatically offloads the loop to GPU and performs necessary data transfers. Use of the \texttt{parallel} loop construct may be sufficient, on its own, to get code running on the GPU, but further directives and clauses exist to give more control to programmer to improve performance and enable more complex cases. By default the compiler will estimate the most efficient decomposition of the loops onto the hierarchical parallelism of the GPU architecture, but in some cases the programmer can achieve better performance by overriding the defaults, and the availability of tuning clauses provide this mechanism. Please see the official documentation for full details.

The programmer can also have more control over data movement. Consider the following combination of two parallel loops:

```fortran
!$acc parallel loop
do i=1, N
  output(i)=2.*input(i)
end do
!$acc end parallel loop
write(*,*) "finished 1st region"

!$acc parallel loop
do i=1, N
  output(i)=i*output(i)
end do
!$acc end parallel loop
```

By default the compiler will unnecessarily copy the output array from and then back to the device between the distinct regions. Such data copies are very expensive so this will result in suboptimal code. The accelerator \texttt{data} construct allows more efficient memory management:

C:

```c
#pragma acc data
{
  ...code region...
}
```

Fortran:

```fortran
!$acc data
  ...code region...
!$acc end data
```

The programmer applies clauses to this construct such as \texttt{copyin} and \texttt{copyout} to specify desired data transfers, for example the above example can be optimised as follows:
Now the only data transfers the copying of the input array in to the device at the start of the data region, and the copying of the output array out of the device at the end of the region. The output array is no longer unnecessarily transferred between the two parallel loops.

We have demonstrated the basic functionality of the accelerator directives programming model. More complex codes require additional functionality, for example the ability to share data between different functions or subroutines. The full set of directives provides support for such cases, please see the official documentation for full details.