Approaches to GPU computing

Manuel Ujaldon
Nvidia CUDA Fellow
Computer Architecture Department
University of Malaga (Spain)
Talk outline [40 slides]

1. Programming choices. [30]
   1. CUDA libraries and tools. [10]
   2. Targeting CUDA to other platforms. [5]
   3. Accessing CUDA from other languages. [4]

2. Examples: Six ways to implement SAXPY on GPUs. [9]

3. Summary. [1]
I. Programming choices
CUDA Parallel Computing Platform

**Programming Approaches**
- Libraries: "Drop-in" Acceleration
- OpenACC Directives: Easily Accelerate Apps
- Programming Languages: Maximum Flexibility

**Development Environment**
- Nsight IDE
  - Linux, Mac and Windows
  - GPU Debugging and Profiling
- CUDA-GDB debugger
  - NVIDIA Visual Profiler

**Open Compiler Tool Chain**
- Enables compiling new languages to CUDA platform, and CUDA languages to other architectures

**Hardware Capabilities**
- SMX
- Dynamic Parallelism
- HyperQ
- GPUDirect
I. 1. CUDA Libraries and tools
Libraries: Easy, high-quality acceleration

Ease of use: Using libraries enables GPU acceleration without in-depth knowledge of GPU programming.

"Drop-in": Many GPU-accelerated libraries follow standard APIs, thus enabling accel. with minimal changes.

Quality: Libraries offer high-quality implementations of functions encountered in a broad range of applications.

Performance: Nvidia libraries are tuned by experts.
Three steps to CUDA-accelerated applications

Step 1: Substitute library calls with equivalent CUDA library calls.

saxpy(...) --> cublasSaxpy (...)

Step 2: Manage data locality.

With CUDA: cudaMalloc(), cudaMemcpy(), etc.
With CUBLAS: cublasAlloc(), cublasSetVector(), etc.

Step 3: Rebuild and link the CUDA-accelerated library.

nvcc myobj.o -l cublas
A linear algebra example

```c
int N = 1 << 20;

// Perform SAXPY on 1M elements: y[] = a*x[] + y[]
saxpy(N, 2.0, x, y, 1);
```
int N = 1 << 20;

// Perform SAXPY on 1M elements: d_y[] = a*d_x[] + d_y[]
cublasSaxpy(N, 2.0, d_x, d_y, 1);

Add "cublas" prefix and use device variables
A linear algebra example

```c
int N = 1 << 20;
cublasInit();

// Perform SAXPY on 1M elements: d_y[] = a*d_x[] + d_y[]
cublasSaxpy(N, 2.0, d_x, d_y, 1);

cublasShutdown();
```

A linear algebra example

```c
int N = 1 << 20;
cublasInit();
cublasAlloc(N, sizeof(float), (void**)&d_x);
cublasAlloc(N, sizeof(float), (void**)&d_y);

// Perform SAXPY on 1M elements: d_y[] = a*d_x[] + d_y[]
cublasSaxpy(N, 2.0, d_x, d_y, 1);

cublasFree(d_x);
cublasFree(x_y);
cublasShutdown();
```

Allocate device vectors

Deallocate device vectors
A linear algebra example

```c
int N = 1 << 20;

// Initialize CUDA libraries

// Allocate memory on GPU

cublasAlloc(N, sizeof(float), (void**)&d_x);
cublasAlloc(N, sizeof(float), (void**)&d_y);

// Set vectors on GPU memory

cublasSetVector(N, sizeof(x[0]), x, 1, d_x, 1);
cublasSetVector(N, sizeof(x[0]), y, 1, d_y, 1);

// Perform SAXPY on 1M elements: d_y[] = a*d_x[] + d_y[]

cublasSaxpy(N, 2.0, d_x, d_y, 1);

// Read data back from GPU memory

cublasGetVector(N, sizeof(y[0]), d_y, 1, y, 1);

cublasFree(d_x);
cublasFree(x_y);

cublasShutdown();
```
CUDA Math Libraries

High performance math routines for your applications:

- **cuFFT**: Fast Fourier Transforms Library.
- **cuBLAS**: Complete BLAS (Basic Linear Algebra Subroutines) Library.
- **cuSPARSE**: Sparse Matrix Library.
- **cuRAND**: RNG (Random Number Generation) Library.
- **NPP**: Performance Primitives for Image & Video Processing.
- **Thrust**: Templated Parallel Algorithms & Data Structures.
- **math.h**: C99 floating-point library.

All included in the CUDA Toolkit. Free download at:

Many other libraries outside the CUDA Toolkit...

- NVIDIA cuBLAS
- NVIDIA cuRAND
- NVIDIA cuSPARSE
- NVIDIA NPP
- GPU VSIPL
- CULA tools
- MAGMA
- NVIDIA cuFFT
- Rogue Wave Software
- ArrayFire
- CUSP
- Thrust

Developed by Nvidia.
Open source libraries.

... not to mention all programs that are available on the Web thanks to the generosity of tough programmers.
Tools and Libraries: Developer ecosystem enables the application growth

- CUDA C/C++
- Parallel Nsight Vis Studio IDE
- NVIDIA Video Libraries
- ParaTools VampirTrace
- PGI Accelerators
- EMPhotonics CULAPACK
- Allinea DDT Debugger
- CUDA X86

- NVIDIA NPP Perf Primitives
- Open CV CUDA Beta
- Bright Cluster Manager
- Thrust C++ Template Lib
- PGI CUDA Fortran
- CAPS HMPP
- MAGMA
- GPU.Net

- pyCUDA
- R-Stream Reservoir Labs
- PBSWorks
- MOAB Adaptive Computing
- Torque Adaptive Computing
- TotalView Debugger
- IMSL
- C++-AMP

- Acceleware EM Library
- Platform LSF Cluster Manager
- TauCUDA Perf Tools
- GPU Packages For R Stats Pkg

Described in detail on Nvidia Developer Zone:

I. 2. Targeting CUDA to other platforms
Compiling for other target platforms
Ocelot
http://code.google.com/p/gpuocelot

It is a dynamic compilation environment for the PTX code on heterogeneous systems, which allows an extensive analysis of the PTX code and its migration to other platforms.

From Feb'11, also considers:
- GPUs manufactured by AMD/ATI.
- CPUs x86 manufactured by Intel.
Swan
http://www.multiscalelab.org/swan

It is a source-to-source translator from CUDA to OpenCL:
- It provides a common API which abstracts the runtime support of CUDA and OpenCL.
- It preserves the convenience of launching CUDA kernels (<<<blocks,threads>>>), generating source C code for the entry point kernel functions.
- ... but the conversion process requires human intervention.

Useful for:
- Evaluate OpenCL performance for an already existing CUDA code.
- Reduce the dependency from nvcc when we compile host code.
- Support multiple CUDA compute capabilities on a single binary.
- As runtime library to manage OpenCL kernels on new developments.
MCUDA
http://impact.crhc.illinois.edu/mcuda.php

Developed by the IMPACT research group at the University of Illinois.

It is a working environment based on Linux which tries to migrate CUDA codes efficiently to multicore CPUs.

Available for free download ...
PGI CUDA x86 compiler
http://www.pgroup.com

Major differences with previous tools:
- It is not a translator from the source code, it works at runtime. It allows to build a unified binary which simplifies the software distribution.

Main advantages:
- **Speed:** The compiled code can run on a x86 platform even without a GPU. This enables the compiler to vectorize code for SSE instructions (128 bits) or the most recent AVX (256 bits).
- **Transparency:** Even those applications which use GPU native resources like texture units will have an identical behavior on CPU and GPU.
- **Availability:** License free for one month if you register as CUDA developer.
I. 3. Accessing CUDA from other languages
Wrappers and interface generators

CUDA can be incorporated into any language that provides a mechanism for calling C/C++. To simplify the process, we can use general-purpose interface generators.

SWIG [http://swig.org] (Simplified Wrapper and Interface Generator) is the most renowned approach in this respect. Actively supported, widely used and already successful with: AllegroCL, C#, CFFI, CHICKEN, CLISP, D, Go language, Guile, Java, Lua, MxScheme/Racket, Ocaml, Octave, Perl, PHP, Python, R, Ruby, Tcl/Tk.

A connection with Matlab interface is also available:
- On a single GPU: Use Jacket, a numerical computing platform.
- On multiple GPUs: Use MatWorks Parallel Computing Toolbox.
Tools available for six different programmer profiles.

1. C programmer  
   CUDA C, OpenACC.

2. Fortran programmer  
   CUDA Fortran, OpenACC.

3. C++ programmer  
   Thrust, CUDA C++.

4. Maths programmer  
   MATLAB, Mathematica, LabVIEW.

5. C# programmer  
   GPU.NET.

6. Python programmer  
   PyCUDA.
Get started today

- These languages are supported on all CUDA GPUs.
- It is very likely that you already have a CUDA capable GPU in your laptop or desktop PC (remember IGPs, EPGs, HPUs).

Web pages:

- **GPU.NET**: [http://tidepowerd.com](http://tidepowerd.com)
- **PyCUDA (Python)**: [http://mathema.tician.de/software/pycuda](http://mathema.tician.de/software/pycuda)
A wild card for languages: On Dec'11, source code of the CUDA compiler was accessible.

This does very convenient and efficient to connect with a whole world of:

- Languages on top. For example, adding front-ends for Java, Python, R, DSLs.
- Hardwares underneath. For example, ARM, FPGA, x86.

CUDA compiler contributed to Open Source LLVM.

CUDA compiler contribu-
NVIDIA GPUs
x86 CPUs
New Processor Support
CU
CUDA
C, C++, Fortran
New language support
LLVM compiler for CUDA

Languages on top. For example, adding front-ends for Java, Python, R, DSLs.

Hardwares underneath. For example, ARM, FPGA, x86.

CUDA compiler contributed to Open Source LLVM.
I. 4. Using directives: OpenACC
OpenACC: A corporative effort for standardization

OpenACC: Open Programming Standard for Parallel Computing

The OpenACC™ API
QUICK REFERENCE GUIDE

The OpenACC Application Program Interface describes a collection of compiler directives to specify loops and regions of code in standard C, C++, and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators.

Most OpenACC directives apply to the immediately following structured block or loop; a structured block is a single statement or a compound statement (C or C++) or a sequence of statements (Fortran) with a single entry point at the top and a single exit at the bottom.

http://www.openacc-standard.org
OpenACC: An alternative to computer scientist’s CUDA for an average programmer

It is a parallel programming standard for accelerators based on directives (like OpenMP), which:

- Are inserted into C, C++ or Fortran programs.
- Drive the compiler to parallelize certain code sections.

Goal: Targeted to an average programmer, code portable across parallel and multicore processors.

Early development and commercial effort:

- The Portland Group (PGI).
- Cray.

First supercomputing customers:

- United States: Oak Ridge National Lab.
- Europe: Swiss National Supercomputing Centre.
Directives provide a **common code base** that is

- Multi-platform.
- Multi-vendor.

This brings an **open** way to preserve investment in legacy applications by enabling an easy **migration** path to accelerated computing.

**GPU directives** allow complete access to the massive parallel **power** of a GPU.

Optimizing code with directives is quite **easy**, especially compared to CPU threads or writing CUDA kernels.

A big achievement is **avoiding restructuring** of existing code for production applications.
OpenACC: How directives work

Starting from simple hints, the compiler parallelizes the code.

It works on:
- Many-core GPUs.
- Multi-core CPUs.

Your original Fortran or C code

Program myscience
... serial code ...
!$acc kernels
  do k = 1,n1
  do i = 1,n2
    ... parallel code ...
    enddo
  enddo
!$acc end kernels
... End Program myscience

OpenACC Compiler Hint
Two basic steps to get started

Step 1: Annotate source code with directives.

```fortran
!$acc data copy util1,util2,util3) copyin(ip,scp2,scp2i)
!$acc parallel loop
... <source code>
!$acc end parallel
!$acc end data
```

Step 2: Compile & run.

```
pgf90 -ta=nvidia -Minfo=accel file.f
```
An example

```fortran
!$acc data copy(A,Anew)
iter=0
do while ( err > tol .and. iter < iter_max )
  iter = iter +1
  err=0._fp_kind

!$acc kernels
  do j=1,m
    do i=1,n
      Anew(i,j) = .25_fp_kind *( A(i+1,j ) + A(i-1,j ) &
                               +A(i ,j-1) + A(i ,j+1))
      err = max( err, Anew(i,j)-A(i,j))
    end do
  end do
!$acc end kernels
  IF (mod(iter,100)==0 .or. iter == 1) print *, iter, err
  A= Anew
end do
!$acc end data
```

- Copy arrays into GPU memory within data region
- Parallelize code inside region
- Close off parallel region
- Close off data region, copy data back
The key question is:
How much performance do we lose?

- Some results say only 5-10% vs. CUDA in "some" cases. Other sources say 5x gains investing a week or even a day.
- But this factor is more application-dependent than influenced by programmer skills.

Real-time object detection
Global Manufacturer of Navigation Systems

Valuation of stock portfolios using Montecarlo
Global Technology Consulting Company

Interaction of solvents and biomolecules
University of Texas at San Antonio

5x in 1 week
2x in 4 hours
5x in 1 day
More recent examples

Lifecycles of fish in Australia

University of Melbourne

Stars and galaxies 12.5B years ago

University of Groningen

Neural networks in self-learning robot

The University of Plymouth

65x in 2 Days

5.6x in 5 Days

4.7x in 4 Hours
A witness from a recent OpenACC workshop at Pittsburgh Supercomputing Center

By end of second day

10x on one atmospheric kernel

6 directives

Technology Director
National Center for Atmospheric Research (NCAR)
More case studies from GTC'13: 3 OpenACC compilers [PGI, Cray and CAPS]

- Performance on M2050 GPU (Fermi, 14x 32 cores), without counting the CPU-GPU transfer overhead.
- Matrix Multiplication size: 2048x2048.
- 7-point Stencil: 3D array size: 256x256x256.

Matrix Multiplication
(Table: # of modified lines, Graph: Performance [Gflops])

<table>
<thead>
<tr>
<th></th>
<th>Baseline</th>
<th>Thread mapping</th>
<th>Cache blocking</th>
<th>Loop unrolling</th>
<th>Shared blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenACC</td>
<td>9</td>
<td>11</td>
<td>62</td>
<td>302</td>
<td>45</td>
</tr>
<tr>
<td>CUDA</td>
<td>26</td>
<td>26</td>
<td>77</td>
<td>317</td>
<td>45</td>
</tr>
</tbody>
</table>

7-Point Stencil
(Table: # of modified lines, Graph: Performance [GB/s])

<table>
<thead>
<tr>
<th></th>
<th>Baseline</th>
<th>Thread mapping</th>
<th>Branch Hoisting</th>
<th>Register blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenACC</td>
<td>7</td>
<td>10</td>
<td>18</td>
<td>29</td>
</tr>
<tr>
<td>CUDA</td>
<td>35</td>
<td>35</td>
<td>45</td>
<td>56</td>
</tr>
</tbody>
</table>

Start now with OpenACC directives

Sign up for a free trial of the directives compiler (thanks to PGI), and get also tools for quick ramp (see http://www.nvidia.com/gpudirectives)

A compiler is also available from CAPS for $199/199€.
II. Programming examples: Six ways to SAXPY on GPUs
What does SAXPY stand for? Single-precision Alpha X Plus Y. It is part of BLAS Library.

```c
void saxpy_serial(float ...)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
```

Using this basic code, we will illustrate six different ways of programming the GPU:

- CUDA C.
- CUBLAS Library.
- CUDA Fortran.
- Thrust C++ Template Library.
- C# with GPU.NET.
- OpenACC.
1. CUDA C

Standard C code:

```c
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
// Invoke SAXPY kernel (serial on 1M elements)
saxpy_serial(4096*256, 2.0, x, y);
```

CUDA code for a parallel execution on GPU:

```c
__global__ void saxpy_parallel(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
// Invoke SAXPY kernel (parallel on 4096 blocks of 256 threads)
saxpy_parallel<<<4096, 256>>>(4096*256, 2.0, x, y);
```
2. CUBLAS Library

Sequential BLAS code

```c
int N = 1 << 20;
// Utiliza la librería BLAS de tu elección
// Invoke SAXPY routine (serial on 1M elements)
blas_saxpy(4096*256, 2.0, x, 1, y, 1);
```

cuBLAS parallel code

```c
int N = 1 << 20;
cublasInit();
cublasSetVector (N, sizeof(x[0]), x, 1, d_x, 1);
cublasSetVector (N, sizeof(y[0]), y, 1, d_y, 1);
// Invoke SAXPY routine (parallel on 1M elements)
cublasSaxpy (N, 2.0, d_x, 1, d_y, 1);
cublasGetVector (N, sizeof(y[0]), d_y, 1, y, 1);
cublasShutdown();
```
## 3. CUDA Fortran

<table>
<thead>
<tr>
<th>Standard Fortran</th>
<th>Parallel Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>module my module contains</td>
<td>module mymodule contains</td>
</tr>
<tr>
<td>subroutine saxpy (n, a, x, y)</td>
<td>attributes(global) subroutine saxpy(n, a, x, y)</td>
</tr>
<tr>
<td>real :: x(:,), y(:,), a</td>
<td>real :: x(:,), y(:,), a</td>
</tr>
<tr>
<td>integer :: n, i</td>
<td>integer :: n, i</td>
</tr>
<tr>
<td>do i=1,n</td>
<td>attributes(value) ::= a, n</td>
</tr>
<tr>
<td>y(i) = a*x(i) + y(i);</td>
<td>i = threadIdx%x + (blockIdx%x-1) * blockDim%x</td>
</tr>
<tr>
<td>enddo</td>
<td>if (i&lt;=n) y(i) = a*x(i) + y(i)</td>
</tr>
<tr>
<td>end subroutine saxpy</td>
<td>end subroutine saxpy</td>
</tr>
<tr>
<td>end module mymodule</td>
<td>end module mymodule</td>
</tr>
</tbody>
</table>

program main
use mymodule
real :: x(2**20), y(2**20)
x = 1.0, y = 2.0

$ Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x, y)

end program main

program main
use cudafor; use mymodule
real, device :: x_d(2**20), y_d(2**20)
x_d = 1.0, y_d = 2.0

$ Perform SAXPY on 1M elements
call saxpy<<<4096,256>>>(2**20, 2.0, x_d, y_d)
y = y_d
end program main
4.1. CUDA C++: Develop Generic Parallel Code

CUDA C++ features enable sophisticated and flexible applications and middleware:

- Class hierarchies.
- __device__ methods.
- Templates.
- Operator overloading.
- Functors (function objects).
- Device-side new/delete.
- ...

```cpp
template<typename T>
struct Functor {
  __device__ Functor(_a) : a(_a) {}  
  __device__ T operator(T x) { return a*x; }  
  T a;
}

template<typename T, typename Oper>
__global__ void kernel(T *output, int n) {
  Oper op(3.7);
  output = new T[n]; // dynamic allocation
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n)
    output[i] = op(i); // apply functor
}
```
4.2. Thrust C++ STL

Thrust is an open source parallel algorithms library which resembles C++ Standard Template Library (STL). Major features:

- **High-level interface:**
  - Enhances developer productivity.
  - Enables performance portability between GPUs and CPUs.

- **Flexible:**
  - CUDA, OpenMP and TBB (Thread Building Blocks) backends.
  - Extensible and customizable.
  - Integrates with existing software.

- **Efficient:**
  - GPU code written without directly writing any CUDA kernel calls.
### Serial C++ Code with STL and Boost

```cpp
int N = 1<<20;
std::vector<float> x(N), y(N);
...

// Invoke SAXPY on 1M elements
std::transform(x.begin(), x.end(), y.begin(), x.end(),
               2.0f * _1 + _2);
```

### Parallel C++ Code

```cpp
int N = 1<<20;
thrust::host_vector<float> x(N), y(N);
...

...  
thrust::device_vector<float> d_x = x;
thrust::device_vector<float> d_y = y;

// Invoke SAXPY on 1M elements
thrust::transform(x.begin(), x.end(), y.begin(), y.begin(),
                   2.0f * _1 + _2);
```

http://www.boost.org/libs/lambda  
http://developer.nvidia.com/thrust
### 5. C# with GPU.NET

<table>
<thead>
<tr>
<th>Standard C#</th>
<th>Parallel C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>private static</td>
<td>[kernel]</td>
</tr>
<tr>
<td>void saxpy (int n, float a, float[] a, float[] y)</td>
<td>private static</td>
</tr>
<tr>
<td>{</td>
<td>void saxpy (int n, float a, float[] a, float[] y)</td>
</tr>
<tr>
<td>for (int i=0; i&lt;n; i++)</td>
<td>{</td>
</tr>
<tr>
<td>y[i] = a*x[i] + y[i];</td>
<td>int i = BlockIndex.x * BlockDimension.x + ThreadIndex.x;</td>
</tr>
<tr>
<td>}</td>
<td>if (i &lt; n)</td>
</tr>
<tr>
<td></td>
<td>y[i] = a*x[i] + y[i];</td>
</tr>
<tr>
<td>int N = 1&lt;&lt;20;</td>
<td>}</td>
</tr>
<tr>
<td>// Invoke SAXPY on 1M elements</td>
<td>int N = 1&lt;&lt;20;</td>
</tr>
<tr>
<td>saxpy(N, 2.0, x, y)</td>
<td>Launcher.SetGridSize(4096);</td>
</tr>
<tr>
<td></td>
<td>Launcher.SetBlockSize(256);</td>
</tr>
<tr>
<td></td>
<td>// Invoke SAXPY on 1M elements</td>
</tr>
<tr>
<td></td>
<td>saxpy(2**20, 2.0, x, y)</td>
</tr>
</tbody>
</table>
# 6. OpenACC Compiler Directives

<table>
<thead>
<tr>
<th>Parallel C Code</th>
<th>Parallel Fortran Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>void saxpy (int n, float a, float[] a, float[] y)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>#pragma acc kernels</td>
<td></td>
</tr>
<tr>
<td>for (int i=0; i&lt;n; i++)</td>
<td></td>
</tr>
<tr>
<td>y[i] = a*x[i] + y[i];</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>// Perform SAXPY on 1M elements</td>
<td></td>
</tr>
<tr>
<td>saxpy(1&lt;&lt;20, 2.0, x, y)</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>subroutine saxpy(n, a, x, y)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>subroutine saxpy(n, a, x, y)</td>
<td></td>
</tr>
<tr>
<td>real :: x(:,), y(:,), a</td>
<td></td>
</tr>
<tr>
<td>integer :: n, i</td>
<td></td>
</tr>
<tr>
<td>!$acc kernels</td>
<td></td>
</tr>
<tr>
<td>do i=1. n</td>
<td></td>
</tr>
<tr>
<td>y(i) = a*x(i) + y(i)</td>
<td></td>
</tr>
<tr>
<td>enddo</td>
<td></td>
</tr>
<tr>
<td>!$acc end kernels</td>
<td></td>
</tr>
<tr>
<td>end subroutine saxpy</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$ Perform SAXPY on 1M elements</td>
<td></td>
</tr>
<tr>
<td>call saxpy(2**20, 2.0, x_d, y_d)</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
There is support for all these 6 approaches on every CUDA GPU (more than 400 million as of 2013). It is very likely that you have one of those within your laptop/desktop.

1. CUDA C/C++

2. CUDA Fortran

3. CUBLAS Library
   http://developer.nvidia.com/cublas

4. Thrust
   http://developer.nvidia.com/thrust

5. C# with GPU.NET
   http://tidepowerd.com

6. OpenACC
   http://developer.nvidia.com/openacc