LINUX VOICE
TUTORIAL

FEEL THE TASTE OF GPU PROGRAMMING

VALENTINE SINITSYN

Use your videocard for non-graphics tasks, and discover a whole new programming paradigm.

WHY DO THIS?

- Make your programs run faster.
- Discover new tools for day-to-day tasks.
- Get prepared for the computing way of tomorrow.

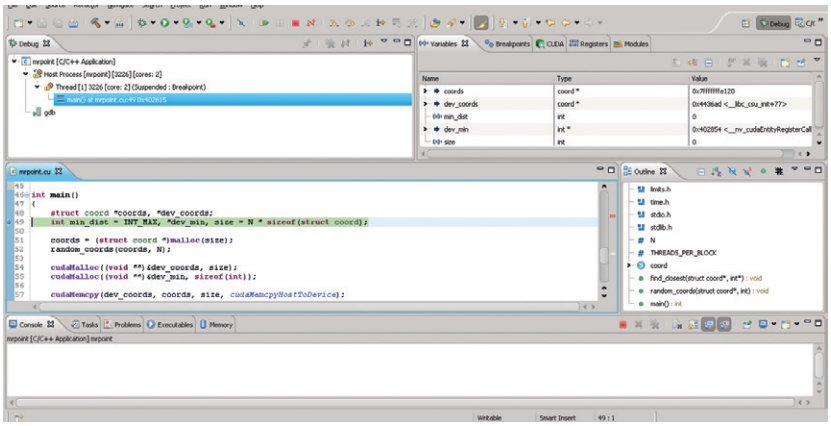
People like faster computers. Faster computers means more numbers to crunch per second, and more importantly, fancier user interfaces and eye candy. For the last decade, many PCs came with videocards delivering decent FPS rates in 3D shooters and enough sides on the *Compiz* cube.

Wouldn't it be cool to have a supercomputer at home? Perhaps you'd be surprised to learn that you already do (almost). Graphics Processing Units (GPUs) on videocards have many (up to thousands) computing cores, come with fast memory, are optimised for number crunching, and are parallel from the ground up. Sounds like a supercomputer to us!

Early days

In the early 2000s, researchers realised that massively parallel GPU architecture works perfect for some scientific problems (eg molecular dynamics). In those days the only available interface to a GPU was OpenGL (or DirectX, for Windows folks). So you needed to express the solution in terms of pixel shaders and texture coordinates. This became known as GP (General Purpose) GPU computing, and this term is sometimes applied to later technologies as well. GPGPU was clever, but neither versatile nor convenient, so Nvidia's CUDA was born in 2007.

CUDA stands for Compute Unified Device Architecture, and it is meant to provide uniform access to Nvidia GPUs (or "devices") for general purpose computations. CUDA programs (by convention, they carry a **.cu** suffix) are written in CUDA C/C++, which is essentially a C language with extensions. You can use other languages as well, but functions to be executed on the GPU (called "kernels" in CUDA parlance; don't confuse them with Linux kernel) always use CUDA C (unless you're from Fortran camp, but we won't discuss that here). Besides the language and compiler for it (LLVM-based **nvcc**), the CUDA Toolkit (**https://developer.nvidia.com/cuda-toolkit**) contains some other tools and a set of libraries, including accelerated BLAS and Sparse BLAS implementations (the *de-facto* standard in scientific computing).



Nsight provides complete Eclipse-based IDE for CUDA programming, including interactive debugger.

CUDA is non-free (as in speech). There is also OpenCL – an open heterogeneous (another term to describe CPU+GPU code) computing specification baked by the Khronos Group. They also maintain OpenGL, and there are certain similarities between these two technologies. Where CUDA relies on language extensions, OpenCL is more like a conventional library with API calls. It's also vendor-neutral: OpenCL is available for AMD, Intel, Nvidia and some others, and supports not only GPUs but also multi-core CPUs and specialised hardware. However, OpenCL implementations aren't necessarily open: say, Beignet (for Intel integrated graphics) is free, while AMD's APP SDK for AMD/ATI videocards isn't.

OpenCL and CUDA coexist peacefully: the former is a way to make ideas behind the latter (historically the first) a formal standard. Nvidia supports it, and many applications come both in CUDA and OpenCL editions.

CUDA and OpenCL provide a two-level hierarchical view on the GPU's computational resources. At the lowest level is the basic entity that executes a kernel – called a "thread" in CUDA or a "worker item" in OpenCL. These are combined in three-dimensional "blocks" or two-dimensional "worker groups". There is an upper limit on the number of threads per block (512 or 1024 in CUDA, depending on your card's age). Finally, blocks and worker groups are combined in a "grid" (CUDA) or "index space" (OpenCL). Many programs use one-dimensional blocks and 1x1 grids, which makes up for a simpler geometry.

A silly example

CUDA and OpenCL make GPUs accessible for general computing, but this approach also has some limitations. First, the amount of RAM available on many videocards is not very large (somewhere in the region of 4GB is pretty common) or easily extensible. Second, early GPUs (prior to CUDA 1.3, or around 2009) lacked support for double-precision floating point arithmetics. Even where available, it's much slower than single-precision. And there are some

tasks that suit the CPU better: GPU computing isn't meant to replace the CPU, but to supplement it.

Say hello to Mister Point. He lives in an unrestricted two-dimensional plane (the poor guy). One day, he comes to his pointy kitchen and spots fire on the curtains. He promptly calls 999 for the fire brigade. As Mr Point resides in highly urbanised area, there are loads of them around, but which one is the closest and quickest to come? This is the question that an emergency phone operator has to answer instantly. As Mr Point is a proud resident of the large city with many blocks, a straight distance between him and a firehouse means next to nothing, and Manhattan norm (the pattern of city blocks) is what we need to consider (ignoring traffic jams for now).

This toy problem demonstrates a typical task that is easy to parallelise. On input, we have a (presumably long) list of fire brigade coordinates (both are pairs of integers to keep things simple). Poor Mr Point is assumed to live at (0, 0). The output is a single integer (the distance to the closest brigade).

Developing a heterogeneous computing program is always about writing kernels and code that launches them, and waiting for the result (besides other things, of course). Let's implement ours for both CUDA and OpenCL, using different host-side languages to feel the difference.

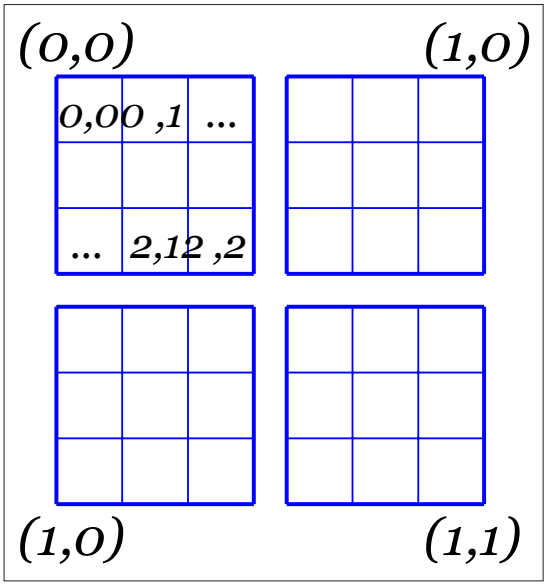
We start with CUDA kernel. For convenience, we declare struct coord (not shown here) which is a pair of integers.

```
#define N (1024 * 1024)
#define THREADS_PER_BLOCK 512

__global__ void find_closest(struct coord *coords, int *closest)
{
 int i = blockDim.x * blockIdx.x + threadIdx.x, t = threadIdx.x;
 __shared__ int dist[THREADS_PER_BLOCK];

 /* Calculate the distance */
 dist[t] = abs(coords[i].x) + abs(coords[i].y);

 __syncthreads();
```



An example of a two-dimensional 3x3 CUDA block aligned in a 2x2 grid.

```
 /* Put minimum distance for this block in dist[0] */
 for (int j = blockDim.x / 2; j > 0; j /= 2) {
          if (t < j && dists[t] > dist[t + j])
            dist[t] = dist[t + j];
   __syncthreads();
 }

 /* Update global minimum distance, if ours is smaller */
 if (t == 0)
   atomicMin(closest, dist[0]);
}
```

__global__ designates the function as a kernel that is callable from the host; there is also the **__device__** attribute for GPU-only visible functions, and **__host__**, which is implicit and is used for the host-only code.

A built-in **threadIdx.[xyz]** variable is the block-local thread identifier. Similarly, **blockIdx.[xyz]** locates that block in a grid, and **blockDim.[xyz]** stores the block size in each dimension. For one-dimensional geometries, only the **.x** part of these variables is used.

Each thread calculates the distance for a given fire brigade and stores the result in the **dist[]** array, which is shared between all threads in a block (hence the **__shared** keyword). Then the array is "reduced" to find the block-local minimum (ie the smallest distance among computed distances in the current block). Before the reduction starts, we must ensure that all threads in a block have calculated the distance and **dist[]** is fully filled. This is what **__syncthreads()** (known as a barrier) is for. We also need a barrier at each reduction iteration to guarantee the array is in consistent state.

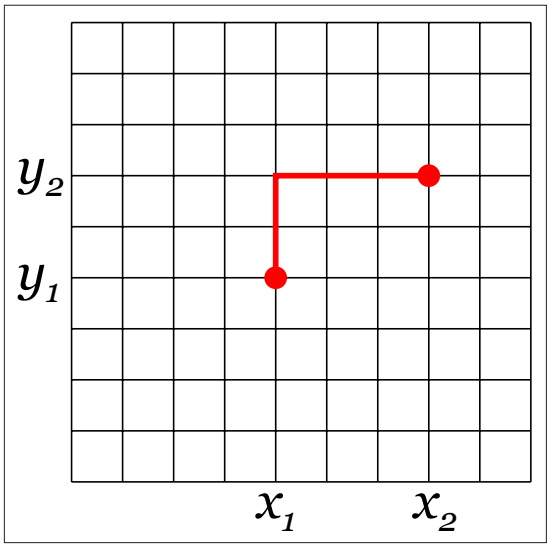
The way we find a block-local minimum isn't straightforward. We are iteratively taking pairs of values and putting the smaller one at a lower index, until the minimal value is written to **dist[0]**. The reason for this complexity is that GPU is a variant of SIMD (Single Instruction – Multiple Data) architecture, and multiple threads (32 in current CUDA devices) are

Obtaining the tools

CUDA is (unfortunately) proprietary, but Nvidia maintains repositories for major distros, including Red Hat and family, SUSE, and Debian/Ubuntu. Download and install the repository configuration files (available as Deb or RPM) then add the software through your package manger.

OpenCL is available from different vendors, and installation methods differ as well. With some, you may be lucky enough to find the required libraries in your distro's package repositories. For others, you may need to download a tarball or an unofficial package.

Either way, pay attention to system requirements. CUDA and OpenCL integrate tightly with host-side tools (**gcc** and alike). Although you may be able to run them on a system that isn't officially supported (I do), I'd recommend you stick to the vendor-approved list for production. Red Hat, SUSE, Debian and friends are usually on it. You're also likely need to install a proprietary graphics driver.



The Manhattan (or taxicab) norm is an easy way to measure distance in a rectangular street grid.

running the same instruction but on different input data. If two threads "diverge", that is, need to run two different instructions (as with many native reduction implementations), it is done in two passes and negatively affects the performance. The algorithm above is the standard way to minimise thread divergence. This being said, the example's code aims at expressiveness, not the speed.

Finally, if block-local minimum is smaller than current 'closest' value, the latter is updated. This way, the smallest per-block distance becomes the result. The first thread in a block (whose **threadIdx.x** is 0) does this, however we can't simply use **if (dist[0] < *closest) *closest = dist[0]** here. The reason is that thread #0 in another block can interleave between the check and the assignment. To prevent the race, one should use atomic operations, like **atomicMin()** above.

Launching kernels

Now, let's turn to the host-side code that launches the kernel. To feel the full taste of CUDA, we'll do it in native C:

```
int main()
{
 struct coord *coords, *dev_coords;
 int min_dist = INT_MAX, *dev_min, size = N * sizeof(struct
coord);

 coords = (struct coord *)malloc(size);
 random_coords(coords, N);

 cudaMalloc((void **)&dev_coords, size);
 cudaMalloc((void **)&dev_min, sizeof(int));

 cudaMemcpy(dev_coords, coords, size,
cudaMemcpyHostToDevice);
 cudaMemcpy(dev_min, &min_dist, sizeof(int),
cudaMemcpyHostToDevice);

 find_closest<<<N/THREADS_PER_BLOCK, THREADS_PER_
BLOCK>>>(dev_coords, dev_min);

 cudaMemcpy(&min_dist, dev_min, sizeof(int),
cudaMemcpyDeviceToHost);
 printf("Mr. Point is at (%d, %d) and the closest brigade is %d
units away\n", 0, 0, min_dist);

 free(coords);
 cudaFree(dev_coords);
 cudaFree(dev_min);

 return 0;
}
```

Launching kernels with CUDA is a three-stage process. First, input data is copied from the host (main RAM) to device (GPU global memory). They are separate memories, and the cost of the copying (although relatively small) should always be kept in mind. Then, the kernel is launched. Finally, the results are copied back from device to host memory.

func<<<N, M>>>(args) is a special syntax for kernel launch. It schedules the kernel on **N** one-dimensional blocks **M** threads each (1x1 grid assumed). There is also an advanced syntax to run kernels on multidimensional blocks and larger grids – consult the CUDA Toolkit Documentation for details. Here, the **1M coords** array is equally split between 2048 blocks 512 threads each. Both values have architecture-defined limits, and you should play with them to see how it affects the performance.

As we are working with C, memory management is manual, and you shouldn't forget to allocate buffers for input and output data and free them when they are no longer needed.

Now, you can compile and run the program with:

```
nvcc -arch sm_20 mrpoint.cu
./mrpoint
Mr. Point is at (0, 0) and the closest brigade is 8 units away
```

It is recommended that you explicitly set the device architecture to match your card's capabilities (CUDA 2.0 here), otherwise you may encounter weird bugs.

The OpenCL way

Let's now see how the same example can be rewritten the OpenCL way. To make things more interesting, we also switch from C to Python for the host-side.

OpenCL vs CUDA

Choosing between OpenCL and CUDA is much like deciding on OpenGL vs DirectX. CUDA is somewhat simpler but Nvidia-only. OpenCL requires more work, but enjoys wider vendor support. The downside of this diversity is that it is harder to optimise your code for each particular device, but you should be able to achieve the same performance with CUDA and OpenCL on the same hardware. There are some discrepancies in feature set (mostly minor), and CUDA has somewhat more advanced tools.

For in-house application targeting Nvidia hardware, we'd probably choose CUDA because of its features and consciousness of API. For a less biased comparison, visit Andreas Klöckner's (the maintainer for both PyCUDA and PyOpenCL) wiki page at **http://wiki.tiker.net/CudaVsOpenCL**.

For OpenCL, the kernel looks almost the same:

```
__kernel void find_closest(__global struct coord *coords, __
global int *closest)
{
 int i = get_global_id(0), t = get_local_id(0);
 __local int dist[THREADS_PER_BLOCK];
 dist[t] = abs(coords[i].x) + abs(coords[i].y);

 barrier(CLK_LOCAL_MEM_FENCE);

 for (int j = get_local_size(0) / 2; j > 0; j /= 2) {
   if (t < j && dist[t] > dist[t + j])
     dist[t] = dist[t + j];
     barrier(CLK_LOCAL_MEM_FENCE);
 }

 if (t == 0)
            atom_min(closest, dist[0]);
}
```

Underscored markers look a bit different, and we need to explicitly say that the arguments come from global memory. Instead of built-in variables, functions are used to get indices (also note that OpenCL provides a direct way for this with no maths involved). **__local** declares worker group shared memory, and **barrier()** creates a barrier (we synchronise local memory access only, as it is where **dist[]** is). Otherwise, the kernel stays pretty the same.

Launching it, however, is more involved process, although PyOpenCL hides some complexity. Compared to CUDA, OpenCL provides no high-level API – that's the price to be paid for flexibility and multivendor support.

```
import numpy as np
import pyopencl as cl

N = (1024 * 1024)
THREADS_PER_BLOCK = 256
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
program = cl.Program(ctx, kernel_src).build()
coords = np.random.randint(-8192, 8192, size=(N, 2)).
astype(np.int32)
min_dist = np.array([2147483647]).astype(np.int32)
mf = cl.mem_flags
dev_coords = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_
PTR, hostbuf=coords)
dev_min = cl.Buffer(ctx, mf.READ_WRITE | mf.COPY_HOST_PTR,
hostbuf=min_dist)
program.find_closest(queue, (N,), (THREADS_PER_BLOCK,),
dev_coords, dev_min)
cl.enqueue_copy(queue, min_dist, dev_min)
print "Mr. Point is at (%d, %d) and the closest brigade is %d units
away" % (0, 0, min_dist)
```

First, we obtain a context encompassing all OpenCL devices in the system. PyOpenCL provides a convenient wrapper for this. We also need a queue to push commands to the OpenCL driver. Then the program is built; **kernel_src** is a string containing its source (you could use string formatting to pass **THREADS_PER_BLOCK** in). There were no stringified sources in CUDA thanks to **nvcc**, but with PyCUDA it would look similar – consider reading external files if it doesn't look neat. PyOpenCL integrates with NumPy, and we use **numpy.array** for data exchange.

To execute a kernel, you call a method on the **program** object. The parameter list contains the global and local sizes and the kernel's arguments. Note that the global size is the total number of worker items in OpenCL (not blocks, as in CUDA), and we could choose to pass **None** later if we wanted the runtime to choose the appropriate local size for us. To get the result, we explictly enqueue the copy operation. Mr Point's trouble was, of course, a simple example. However, with some generalisations it may form a building block for a more complex task like classification or character recognition.

There are specialised massively parallel accelerator boards, like Intel Xeon Phi, or Nvidia Tesla found in this author's new server. They are fully supported by CUDA and/or OpenCL.

Everyone's covered

At this point you may think: "GPU's benefits for scientific computing are clear, but I'm not into it, so why should I care?" Glad you asked. While the APIs certainly target writers of high-performance code, there are tools ready that are useful for non-programmers as well.

Administrators can secure their networks with *Suricata* IDS/IPS (**http://suricata-ids.org**), which uses CUDA to speed up protocol, file etc detection in network traffic. You still need a decent network card to capture packets quickly, but GPU processing will help you to discover potential threats faster. There are also many WPA/ZIP file/whatever else password recovery utilities: a tool like *Hashcat* (**http://hashcat.net**) would certainly have improved your chances of winning the LV's Password Cracking Challenge. There are other legitimate uses for these tools but keep in mind that are in the bad guys' arsenal as well, and don't forget to use strong passwords (run some of these password crackers on your password file to see if you are already in danger).

GPU computing has many other applications in medicine, engineering and even finance, and we'll certainly see more in the future. Stay tuned!

Dr Valentine Sinitsyn spends half of the day in university where he teaches students physics and diagonalizes large matrices.