# 45

# Introduction to GPGPU for AI

*Conan Bourke and Tomasz Bednarz*



## 45.1 Introduction

Computer hardware has come a long way in the past decade. One core, two core, four core, and now hundreds and thousands of cores! The power in a computer has shifted from the CPU to the GPU, with new APIs allowing programmers to take control of these chips for more than just graphics processing.

With each advance in hardware, game systems have gained more and more processor power, allowing for far more intricate and detailed experiences than previously imagined. For AI this has meant more realistic agents capable of far more complex interactions with each other, the player, and the environment around them.

One of the most important advancements in hardware has been in the GPU and its transition from serving purely as a rendering processor into a general floating-point processor capable of any calculation we wish, within certain limits. The two major hardware vendors, AMD and NVIDIA, both make GPUs that commonly have 512+ processors, with the newest models providing around 3000 processors in consumer models, each able to process data in parallel. That is a lot of processing power, and we don't have to dedicate all of it to high-end graphics. Even the Xbox 360 has a GPU capable of basic general processing techniques, and Sony's PS3 has an architecture similar to the GPU that is capable of the same sort of processing.

One common downside to these processors is the latency and bandwidth between CPU and GPU, but inroads are being made, particularly by AMD, in combining the linear computation models of GPUs with the generic processing models of CPUs, called Accelerated

Processing Units (APUs), that greatly reduce this latency and provide other advantages as will be discussed later in the chapter.

## 45.2 A History of GPGPU

General-purpose computation on the GPU (GPGPU) is the term given to using the GPU for calculations other than rendering [Harris 02]. With the introduction of early shader models, programmers were able to modify how and what the GPU processed, no longer having to rely on the fixed-function pipelines of OpenGL or Direct3D and not being forced to render out an image for viewing. Instead textures could be used as buffers of data, accessed and processed within a pixel fragment, and the result could be drawn to an output texture. This method had the drawback that buffers were read-only or write-only. In addition, limitations with element independence and the need to represent problems in the context of graphics algorithms and the render pipeline made this technique cumbersome. Nevertheless shaders had evolved from simple assembly language programs into several new C-like languages, and hardware vendors began to acknowledge the growing GPGPU field.

Work was done on exposing the capabilities of the GPU outside of the rendering pipeline, and thus APIs were created that gave programmers access to the power of the GPU without having to treat it as a purely graphics-based device, and with these APIs came buffers that had read-write-modify access and additional mathematical capabilities.

In February 2007 NVIDIA introduced the Compute Unified Device Architecture (CUDA) API with the G80 series of GPUs which heavily accelerated the GPGPU field, and with DirectX11 Microsoft released the DirectCompute API. It shares many similar ideas and methodologies to CUDA but with the advantage of using DirectX's existing resources, allowing easy sharing of data between DirectCompute and Direct3D for visualizations. DirectCompute is also usable on DirectX10 hardware, but is still limited with the rest of DirectX in that it can only run on Windows-based systems.

A competing standard, OpenCL (Open Computing Language), was first released in 2008. OpenCL gives developers easy access to write efficient cross-platform applications for heterogeneous architectures such as multicore CPUs and GPUs, even Sony's PS3, with a single programming interface based on a modern contemporary of the C language. OpenCL's specification is managed by the Khronos Group [Khronos] who also provide a set of C++ bindings we have used for the purpose of this article. These bindings greatly simplify the host API setup and speed of code development.

## 45.3 OpenCL

OpenCL programs are written as "kernels," functions that execute on single processing units (compute units) in parallel to the other processing units, working independently from each other. Kernel code is very similar to C code, and supports many built-in math functions.

When a host program is running we need to execute the following steps to use OpenCL:

1. OpenCL enumerates the platforms and the compute devices available in the system, i.e., all CPUs and GPUs. Within each device there are one or more compute units, and within these one or more processing units that handle the actual computation.

The number of units corresponds to the number of independent instructions that a device (CPU or GPU) can execute at the same time. Therefore a CPU having 8 units is still considered as a single device, as would be a GPU with 448 units.

2. The most powerful device (usually the GPU) is picked, and its context is set up for further operations.
3. Sharing of data is configured between the host application and OpenCL.
4. An OpenCL program is built for your device using kernel code and an OpenCL context, and then kernel objects are extracted from compiled kernel code.
5. OpenCL makes use of command queues to control synchronization of kernel executions. Reading and writing data to and from the kernel and manipulation of memory objects are also carried out by the command queues.
6. The kernel is invoked and executes across all processing units. The parallel threads share memory and synchronize using barriers. At the end eventually, the output of the work-items is read back into host memory for further manipulation.

## 45.4 Simple Flocking and OpenCL

What we will cover briefly is the conversion of a classic AI algorithm to run on the GPU using OpenCL. Craig Reynolds [Reynolds 87] introduced the concept of *steering behaviors* for controlling autonomous moving agents and with it the concept of *flocking* and *boids* to simulate crowds and flocks. Many RTS games make use of flocking with beautiful effect—Relic Entertainment's *Homeworld* series being one such example—but these games are usually limited in the number of agents available. Converting this algorithm to run on the GPU, we can increase our agent counts into the thousands quite easily.

We will implement a brute-force approach to flocking, on both the CPU and GPU, to demonstrate the easy gains to be had by simply switching to the GPU without utilizing any partitioning, taking advantage of the GPUs massively parallel architecture. Similar work has been done using the PS3's Cell Architecture [Reynolds 06] utilizing a simple spatial partitioning scheme. Listing 45.1 gives pseudocode for a basic flocking algorithm using a prioritized weighted sum of forces for separation, cohesion, and alignment, and also includes a wander behavior to help randomize the agents. The priority is wander, and then separation, cohesion, and finally alignment. All velocities must be updated before being applied to positions or initial agents will incorrectly influence later agents.

**Listing 45.1** Flocking pseudocode.

```
for each agent
    for each neighbor within radius
        calculate agent separation force away from neighbor
        calculate center of mass for agent cohesion force
        calculate average heading for agent alignment force
    calculate wander force
    sum prioritized weighted forces and apply to agent velocity
for each agent
    apply velocity to position
```

On the CPU this algorithm is trivial to implement. Usually an agent will consist of an object containing the relevant agent information (i.e., position, velocity, wander target, and so on). An array of agents would then be looped over twice; first to update the forces and velocity for each agent, and second to apply the new velocity.

When converting this algorithm to the GPU there are a few items that must be considered, but the conversion of CPU code to OpenCL code is straightforward. As seen in the pseudocode above, all neighbors are calculated for each agent, which has $0(n^2)$ complexity. On the CPU this is achieved by double looping through all the agents. On the GPU we are able to parallelize the outer loop and execute sequentially the inner loop interaction for every work item (every agent), greatly reducing the processing time.

Spatial partitioning techniques could be implemented to increase performance, but it must be noted that the GPU works in a very linear fashion, ideal for processing arrays of data which it typically did in the case of vertex arrays for graphics processing. In the case of complex spatial partitioning schemes (such as an octree) the GPU would flail while trying to access nonlinear memory. Craig Reynolds' solution for the PS3 was to use a simple three-dimensional grid of buckets storing neighboring agents [Reynolds 06]. This allows buckets to be processed linearly, with agents only having to have read access to the buckets directly neighboring their own. With this article, however, we are demonstrating a simple conversion from CPU to GPU without this kind of optimization to show the instant gains from converting to GPGPU processing.

One of the first steps when converting to GPGPU is to break up your data into contiguous arrays. GPUs can handle up to three-dimensional arrays, but in our example we will break up our agents into one-dimensional arrays for each element in an agent, that is, positions, velocities, etc.

It is also worth noting that in OpenCL terminology there are two types of memory: local and global. The distinction is that global memory can be accessed by any core, while local memory is unique to a process and is therefore accessed much faster. Think of it like RAM and a CPU's Cache.

## 45.5 OpenCL Setup

Initializing the compute devices is straightforward using C++ host bindings. At first the host platforms have to be enumerated to access the underlying compute devices. Then a context is created from a platform (note that in this example we initialized the context to specifically use the GPU using CL_DEVICE_TYPE_GPU) along with a command queue in order to execute compute kernels and enqueue memory transfers via the context. Refer to Listing 45.2 for the details.

OpenCL has two types of memory objects: buffers and images. Buffers contain standard 4D floating-point vectors using a Single-Instruction Multiple-Data (SIMD) processing model, while images are defined in terms of texels. For the purposes of this article buffers were chosen as being more appropriate for representing agents located contiguously beside each other.

The buffers can be initialized for read-only, write-only, or read–write, as Listing 45.3 shows. The buffers were created to hold the maximum numbers of agents the simulation will make use of, though we are able to process fewer agents if we desire. In addition to

Listing 45.2. OpenCL host setup.

```
cl::Platform::get(&m_oclPlatforms);
cl_context_properties aoProperties[] = {
    CL_CONTEXT_PLATFORM,
    (cl_context_properties)(m_oclPlatforms[0])(),
    0};
m_oclContext = cl::Context(CL_DEVICE_TYPE_GPU, aoProperties);
m_oclDevices = m_oclContext.getInfo<CL_CONTEXT_DEVICES>();
std::cout << "OpenCL device count: " << m_oclDevices.size();
m_oclQueue = cl::CommandQueue(m_oclContext, m_oclDevices[0]);
```

Listing 45.3 OpenCL buffer setup.

```
typedef struct Params
{
    float fNeighborRadiusSqr;
    float fMaxSteeringForce;
    float fMaxBoidSpeed;
    float fWanderRadius;
    float fWanderJitter;
    float fWanderDistance;
    float fSeparationWeight;
    float fCohesionWeight;
    float fAlignmentWeight;
    float fDeltaTime;
} Params;
cl::Buffer m_clVPosition;
cl::Buffer m_clVVelocity;
cl::Buffer m_clVParams;
...
m_clVPosition = cl::Buffer(m_oclContext, CL_MEM_READ_WRITE,
uiMaxAgentCount * 4 * sizeof(float));
m_clVParams = cl::Buffer(m_oclContext, CL_MEM_READ_ONLY,
sizeof(Params));
```

agent data we send to the kernel the parameters for the flocking algorithm, along with a time value specifying elapsed time since the last frame for consistent velocities.

In order to create a compute kernel we need to compile the kernel code into a CL program, and then extract the compute kernel. In our example the kernel code is located in a separate file `program.cl`, loaded to create the program, as shown in Listing 45.4.

Listing 45.5 shows a portion of our example kernel, with the body omitted as it is nearly identical to a CPU implementation. Of note, however, is the last portion of the kernel pertaining to barriers. On the CPU we loop twice to apply the forces to all agents after they have been calculated. We can achieve this in the kernel by placing a barrier, which causes all executed threads to wait at this point until all threads have caught up. Within a kernel

Listing 45.4 Building the OpenCL program.

```
//read source file
std::ifstream sFile("program.cl");
std::string sCode(std::istreambuf_iterator<char>(sFile),
    (std::istreambuf_iterator<char>()));
cl::Program::Sources oSource(1,
    std::make_pair(sCode.c_str(), sCode.length() + 1));
//build the program for the specified devices
m_oclProgram = cl::Program(m_oclContext, oSource);
m_oclProgram.build(m_oclDevices);
m_clKernel = cl::Kernel(m_oclProgram, "Flocking");
```

Listing 45.5. The OpenCL kernel.

```
__kernel void Flocking(
        __global float4* vPosition,
        ...
        __constant struct Params* pp)
{
    //get_global_id(0) accesses the current element index
    unsigned int i = get_global_id(0);
    ...
    barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
    vPosition[i] + = vVelocity[i] * pp->fDeltaTime;
    barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
}
```

Listing 45.6. Specifying kernel arguments.

```
m_clKernel.setArg(0, sizeof(cl_mem), &m_clVPosition);
m_clKernel.setArg(1, sizeof(cl_mem), &m_clVVelocity);
```

we can access the current index of the input buffers with a call to `get_global_id(0)` using 0, 1, or 2 depending on the buffer dimensions.

Kernel arguments are passed to OpenCL explicitly once a kernel has been built, shown in listing 45.6. Rather than passing the arguments to the kernel when it is executed, the arguments must be pre-loaded into their corresponding argument index.

Once everything has been initialized and built, we can enqueue our kernel to be computed. Kernels do not execute immediately but are rather queued up to be processed. The kernel must be launched with global work-size equal to the number of elements to be processed. We can also specify an offset into our array range, but we can specify a NullRange to start at the front of the array. Refer to Listing 45.7.

Listing 45.7. Executing the kernel program.

```
m_oclQueue.enqueueNDRangeKernel(
    m_clKernel, cl::NullRange,
    cl::NDRange(uiMaxAgentCount),
    cl::NullRange,
    nullptr, nullptr);
```

## 45.6 Sharing GPU Processing Between Systems

Initial concerns developers may have when making use of GPGPU, especially for game developers, is that processing time is taken away from graphics processing. Many of today's high-end games make use of GPGPU for graphical pre-processing and post-processing, and also for physics simulations using APIs such as NVidia's PhysX. Adding AI to the mix will reduce the processing time these other systems have available. This is a concern that cannot be avoided. However GPU processing power has increased with massive leaps and bounds, from the core counts in the hundreds for the NVidia 500 series, to thousands in their 600 series. With time more processing power will be available for more systems, and developers will start to find other interesting uses for that power besides graphics, physics, and AI.

In the meantime, at least for OpenCL, there exists interoperability APIs that allow sharing of OpenCL buffers between both OpenGL and Direct3D, reducing the need to constantly transfer information to the GPU and back to the CPU. Positional buffers for AI agents could be both used in flocking computations on the GPU—for instance, rendering buffers for hardware instancing of rendered agents, without the need to return the data to the CPU only for it to be transferred back to the GPU.

## 45.7 Results

Figure 45.1 displays performance measured in milliseconds to process agents with the brute-force implementation of our example flocking algorithm (lower is better). As clearly shown the GPU offers a massive performance increase with higher agent counts, with minimal work needed to convert the algorithm to OpenCL. However, at lower agent counts the GPU runs slower than the CPU, shown in Figure 45.2, due to the buffer transfers. Also tested is a GPU intended for computation and research, to show the optimized bandwidth and latency in such devices, resulting in faster computations than consumer level GPUs, but also giving an insight into future consumer level performance.

## 45.8 Conclusion

As our example shows we could easily use GPGPU computing for a game that makes use of extremely high agent counts, such as an RTS game using thousands of entities rather than just the standard dozens to hundreds in most current RTS games. We would still have difficulty moving our agent's decision making to the GPU, but elements such as locomotion

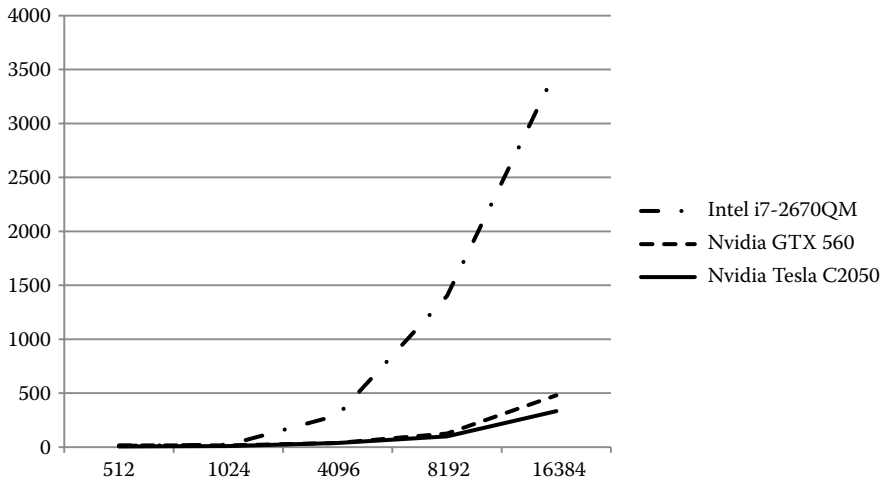


Figure 45.1

Performance in milliseconds to process agent counts of 512, 1024, 4096, 8192, and 16384 with various GPUs and CPUs.

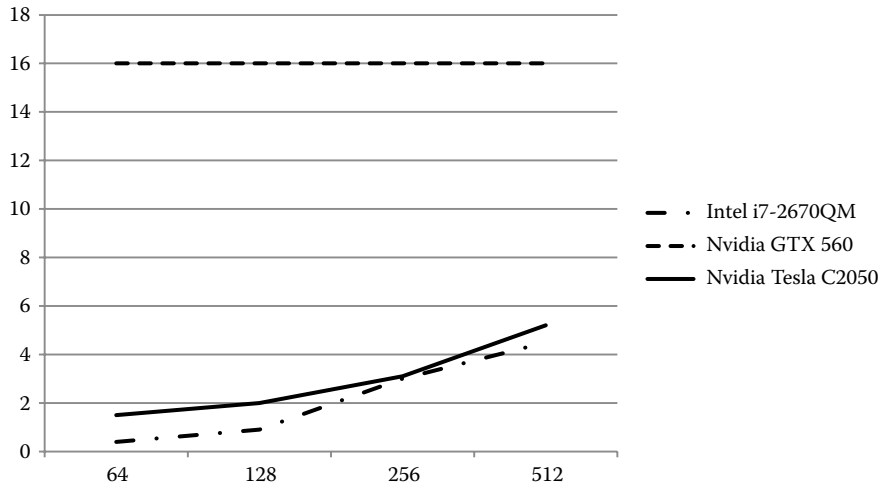


Figure 45.2

Performance in milliseconds with lower agent counts. The consumer GPU stays constant as the processing of the agents takes next to no time at all but buffer transfers to and from the GPU are not optimized. However, the C2050 has a much higher bandwidth and latency.

and even obstacle avoidance can be moved off the CPU. Taking processing time away from other systems, such as graphics, will be another concern, but can be alleviated somewhat with various interoperability APIs.

Other examples of GPGPU for AI exist, and work has been done by groups for neural networks and pathfinding, and the classic Conway's Game of Life can easily be

implemented on the GPU [Rumpf 10]. The main limit to the types of AI processing available in GPGPU is the branching nature of most AI decision making.

APUs could allow us to closely couple decision making techniques with GPGPU techniques, but the consumer take-up for such devices will dictate if this style of AI will show up more in games.

As it stands GPGPU is a viable option for mass simulations, with the current most common consumer GPU at the time of writing, according to Valve's Hardware Survey, being the NVidia GTX 560 which packs 336 processors. Plenty for our AI needs.

## References


[Harris 02] M. Harris. GPGPU.org. http://www.gpgpu.org, 2002.

[Khronos] The Khronos Group. http://www.khronos.org.

[Reynolds 87] C. Reynolds. "Flocks, herds, and schools: A distributed behavioral model." *SIGGRAPH '87 Conference Proceedings*. Available online (http://www.red3d.com/cwr/papers/1987/SIGGRAPH87.pdf).

[Reynolds 06] C. Reynolds. "Big fast crowds on PS3." *SIGGRAPH '06 Sandbox Symposium*. Available online (http://www.research.scea.com/pscrowd/PSCrowdSandbox2006.pdf).

[Rumpf 10] T. Rumpf. "Conway's game of life accelerated with OpenCL." *CMC11 '10 Conference Proceedings*. Available online (http://cmc11.uni-jena.de/proceedings/rumpf.pdf).