General Purpose GPU Computing

So far, our coverage of parallel programming has been partial to CPU cores. Indeed, we have several skills at our disposal to parallelize programs across multiple cores, synchronize access to shared resources, and use high-speed CPU primitives for lock-free synchronization. As we noted at the outset of this chapter, there is another source of parallelism available to our programs—the GPU, which on modern hardware offers considerably many cores than even high-end CPUs. GPU cores are very suitable for data-parallel algorithms, and their sheer number makes up for the clumsiness often associated with running programs on them. In this section we examine one way of running programs on the GPU, using a set of C++ language extensions called C++ AMP.

Note C++ AMP is based on C++, which is why this section will use C++ code examples. However, by applying a moderate amount of .NET Interoperability, you can use C++ AMP algorithms in your .NET applications as well. We will return to this subject at the end of the section.

Introduction to C++ AMP

At its essence, a GPU is a processor like any other, with a specific instruction set, numerous cores, and a memory access protocol. However, there are significant differences between modern GPUs and CPUs, and understanding them is central for writing efficient GPU-based programs:

- There is only a small subset of instructions available to modern CPUs that are available on the GPU. This implies some limitations: there are no function calls, data types are limited, library functions are missing, and others. Other operations, such as branching, may introduce a performance cost unparalleled to that on the CPU. Clearly, this makes porting massive amounts of code from the CPU to the GPU a considerable effort.
- There is a considerably larger number of cores on a mid-range graphics card than on a mid-range CPU socket. There will be work units that are too small or cannot be broken into sufficiently many pieces to benefit properly from parallelization on the GPU.
- There is scarce support for synchronization between GPU cores executing a task, and no support for synchronization between GPU cores executing different tasks. This requires the synchronization and orchestration of GPU work to be performed on the CPU.

WHAT TASKS ARE SUITABLE FOR EXECUTION ON THE GPU?

Not every algorithm is suitable for execution on the GPU. For example, GPUs do not have access to other I/O devices, so you can scarcely improve the performance of a program that fetches RSS feeds from the Web by using a GPU. However, many CPU-bound data-parallel algorithms can be ported to the GPU and procure from it massive parallelization. Here are some examples (the list is by no means exhaustive):

- Image blur, sharpen, and other transformations
- Fast Fourier Transform
- Matrix transpose and multiplication
- Number sorting
- Brute-force hash reversal

A good source of additional examples is the Microsoft Native Concurrency team blog (`http://blogs.msdn.com/b/nativeconcurrency/`), which has sample code and explanations for a variety of algorithms that have been ported to C++ AMP.

C++ AMP is a framework that ships with Visual Studio 2012 and provides C++ developers with simple means to run computations on the GPU, and requires only a DirectX 11 driver to run. Microsoft has released C++ AMP as an open specification (available online at the time of writing at `http://blogs.msdn.com/b/nativeconcurrency/archive/2012/02/03/c-amp-open-spec-published.aspx`), which any compiler vendor can implement. In C++ AMP, code can execute on *accelerators*, which represent computational devices. C++ AMP discovers dynamically all the accelerators with a DirectX 11 driver. Out of the box, C++ AMP also ships with a reference accelerator that performs software emulation, and a CPU-based accelerator, WARP, which is a reasonable fallback on machines without a GPU or with a GPU that doesn't have a DirectX 11 driver, and uses multi-core and SIMD instructions.

With no further ado, let us consider an algorithm that can be easily parallelized on the GPU. The algorithm below takes two vectors of the same length and calculates a pointwise result. There's hardly anything that could be more straightforward:

```
void VectorAddExpPointwise(float* first, float* second, float* result, int length) {
  for (int i = 0; i < length; ++i) {
    result[i] = first[i] + exp(second[i]);
  }
}
```

Parallelizing this algorithm on the *CPU* requires splitting the iteration range into several chunks and creating a thread to handle each part. Indeed, we have devoted quite some time to doing just that with our primality testing example—we have seen how to parallelize it by manually creating threads, by issuing work items to the thread pool, and by using the `Parallel.For` automatic parallelization capabilities. Also, recall that when parallelizing similar algorithms on the CPU we have taken great care to avoid work items that were too granular (e.g., a work item per iteration wouldn't do).

On the *GPU*, no such caution is necessary. The GPU is equipped with many cores that can execute threads very rapidly, and the cost of a context switch is significantly lower than on the CPU. Below is the code required using C++ AMP's `parallel_foreach` API:

```
#include < amp.h>
#include < amp_math.h>
using namespace concurrency;
void VectorAddExpPointwise(float* first, float* second, float* result, int length) {
  array_view < const float,1 > avFirst (length, first);
  array_view < const float,1 > avSecond(length, second);
  array_view < float,1> avResult(length, result);
  avResult.discard_data();
  parallel_for_each(avResult.extent, [=](index < 1 > i) restrict(amp) {
    avResult[i] = avFirst[i] + fast_math::exp(avSecond[i]);
  });
```

```
    avResult.synchronize();
  }
```

We now examine each part of the code individually. First, the general shape of the main loop has been maintained, although the original `for` loop has been replaced with an API call to `parallel_foreach`. Indeed, the principle of converting a loop to an API call is not new—we have seen the same with TPL's `Parallel.For` and `Parallel.ForEach` APIs.

Next, the original data passed to the method (the `first`, `second`, and `result` parameters) has been wrapped in `array_view` instances. The `array_view` class wraps data that must be moved to an accelerator (GPU). Its template parameters are the type of the data and its dimensionality. If we want the GPU to execute instructions that access data that was originally on the CPU, some entity must take care of copying the data to the GPU, because most of today's GPUs are discrete devices with their own memory. This is the task of the `array_view` instances—they make sure data is copied on demand, and only when it is required.

When the work on the GPU is done, the data is copied back to its original location. By creating `array_view` instances with a `const` template type argument, we make sure that `first` and `second` are copied only *to* the GPU but don't have to be copied back *from* the GPU. Similarly, by calling the `discard_data` method, we make sure that `result` is not copied from the CPU to the GPU, but only from the GPU to the CPU when there is a result worth copying.

The `parallel_foreach` API takes an *extent*, which is the shape of the data we are working on, and a function to execute for each of the elements in the extent. We used a lambda function in the code above, which is a welcome addition to C++ as of the 2011 ISO C++ standard (C++11). The `restrict(amp)` keyword instructs the compiler to verify that the body of the function can execute on the GPU, forbidding most of the C++ syntax—syntax that cannot be compiled to GPU instructions.

The lambda function's parameter is an `index < 1 >` object, which represents a one-dimensional index. This must match the extent we used—should we declare a two-dimensional extent (such as the data shape of a matrix), the index would have to be two-dimensional as well. We will see an example of this shortly.

Finally, the `synchronize` method call at the end of the method makes sure that by the time `VectorAdd` returns, the changes made on the CPU to the `avResult array_view` are copied back into its original container, the `result` array.

This concludes our first foray into the world of C++ AMP, and we are ready for a closer examination of what is going on—as well as a better example, which will yield benefits from GPU parallelization. Vector addition is not the most exciting of algorithms, and does not make a good candidate for offloading to the GPU because the memory transfer outweights the computation's parallelization. In the following subsection we look at two examples that should be more interesting.

Matrix Multiplication

The first "real-world" example we shall consider is matrix multiplication. We will optimize the naïve cubic time algorithm for matrix multiplication and not Strassen's algorithm, which runs in sub-cubic time. Given two matrices of suitable dimensions, A that is m-by-w and B that is w-by-n, the following sequential program produces their product, a matrix C that is m-by-n:

```
void MatrixMultiply(int* A, int m, int w, int* B, int n, int* C) {
  for (int i = 0; i < m; ++i) {
    for (int j = 0; j < n; ++j) {
    int sum = 0;
    for (int k = 0; k < w; ++k) {
    sum + = A[i*w + k] * B[k*w + j];
    }
    C[i*n + j] = sum;
    }
  }
}
```

There are several sources for parallelism here, and if you were willing to parallelize this code on the CPU, you might be right in suggesting that we parallelize the outer loop and be done with it. On the GPU, however, there are sufficiently many cores that if we parallelize only the outer loop we might not create enough work for all the cores. Therefore, it makes sense to parallelize the two outer loops, still leaving a meaty algorithm for the inner loop:

```
void MatrixMultiply(int* A, int m, int w, int* B, int n, int* C) {
  array_view < const int,2 > avA(m, w, A);
  array_view < const int,2 > avB(w, n, B);
  array_view < int,2> avC(m, n, C);
  avC.discard_data();
  parallel_for_each(avC.extent, [=](index < 2 > idx) restrict(amp) {
    int sum = 0;
    for (int k = 0; k < w; ++k) {
    sum + = avA(idx[0]*w, k) * avB(k*w, idx[1]);
    }
    avC[idx] = sum;
  });
}
```

Everything is still very similar to sequential multiplication and the vector addition example we have seen earlier—with the exception of the index, which is two-dimensional, and accessed by the inner loop using the `[]` operator. How does this version fare compared to the sequential CPU alternative? To multiply two 1024 × 1024 matrices (of integers), the CPU version required 7350 ms on average, whereas the GPU version—hold tight—took 50 ms on average, a *147-fold improvement*!

N-Body Simulation

The examples we have seen so far had very trivial code in the inner loop that was scheduled on the GPU. Clearly, this must not always be the case. One of the examples on the Native Concurrency team blog we referred to demonstrates an N-body simulation, which simulates the interactions between particles where the force of gravity is applied. The simulation consists of an infinite number of steps; in each step, it has to determine the updated acceleration vector of each particle and then determine its new location. The parallelizable component here is the vector of particles—with sufficiently many particles (a few thousands or more) there is ample work for all GPU cores to do at once.

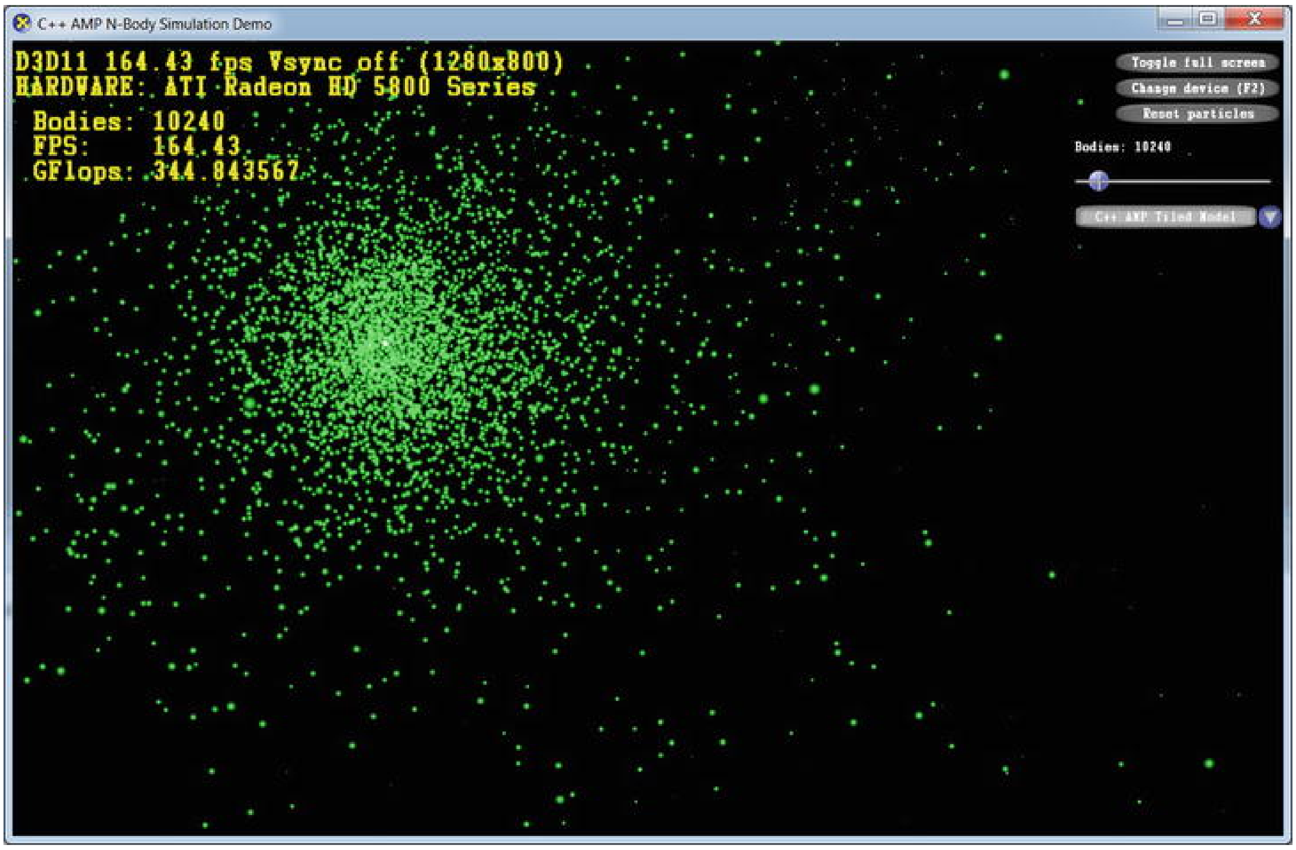
The kernel that determines the result of an interaction between two bodies is the following code, which can be ported very easily to the GPU:

```
//float4 here is a four-component vector with pointwise operations
void bodybody_interaction(
  float4& acceleration, const float4 p1, const float4 p2) restrict(amp) {
  float4 dist = p2 – p1;
  float absDist = dist.x*dist.x + dist.y*dist.y + dist.z*dist.z; //w is unused here
  float invDist = 1.0f / sqrt(absDist);
  float invDistCube = invDist*invDist*invDist;
  acceleration + = dist*PARTICLE_MASS*invDistCube;
}
```

Each simulation step takes an array of particle positions and velocities, and generates a new array of particle positions and velocities based on the results of the simulation:

```
struct particle {
  float4 position, velocity;
  //ctor, copy ctor, and operator = with restrict(amp) omitted for brevity
};
void simulation_step(array < particle,1 > & previous, array < particle,1 > & next, int bodies)
  extent < 1 > ext(bodies);
  parallel_for_each(ext, [&](index < 1 > idx) restrict(amp) {
    particle p = previous[idx];
    float4 acceleration(0, 0, 0, 0);
    for (int body = 0; body < bodies; ++body) {
    bodybody_interaction(acceleration, p.position, previous[body].position);
    }
    p.velocity + = acceleration*DELTA_TIME;
    p.position + = p.velocity*DELTA_TIME;
    next[idx] = p;
  });
}
```

With an appropriate GUI, this simulation is very entertaining. The full sample provided by the C++ AMP team is available on the Native Concurrency blog. On the author's system, an Intel i7 processor with an ATI Radeon HD 5800 graphics card, a simulation with 10,000 particles yielded ~ 2.5 frames per second (steps) from the sequential CPU version and 160 frames per second (steps) from the optimized GPU version (see Figure 6-11), an incredible improvement.



```
D3D11 164.43 fps Vsync off (1280x800)
HARDWARE: ATI Radeon HD 5800 Series
 Bodies: 10240
 FPS:    164.43
 GFlops: 344.843567
```

Figure 6-11 . N-body simulation UI demo, showing >160 frames per second (simulation steps) when using the optimized C++ AMP implementation with 10,240 simulated particles

Tiles and Shared Memory

Before we conclude this section, there is a very important optimization surfaced by C++ AMP that can improve even more the performance of our GPU code. GPUs offer a programmable data cache (often called *shared memory*). Values stored in it are shared across all threads in the same *tile*. By using tiled memory, C++ AMP programs can read data from the GPU's main memory once into the shared tile memory, and then access it quickly from *multiple* threads in the same tile without refetching it from the GPU's main memory. Accessing shared tile memory can be around 10 times faster than accessing the GPU's main memory—in other words, you have a reason to keep reading.

To execute a tiled version of a parallel loop, the `parallel_for_each` method accepts a `tiled_extent` domain, which subdivides a multi-dimensional extent into multi-dimensional tiles, and a `tiled_index` lambda parameter, which specifies both the global thread ID within the extent, and the local thread ID within the tile. For example, a 16 × 16 matrix can be subdivided into 2 × 2 tiles (see Figure 6-12) and then passed into `parallel_for_each`:

```
extent < 2 > matrix(16,16);
tiled_extent < 2,2 > tiledMatrix = matrix.tile < 2,2 > ();

parallel_for_each(tiledMatrix, [=](tiled_index < 2,2 > idx) restrict(amp) { . . . });
```

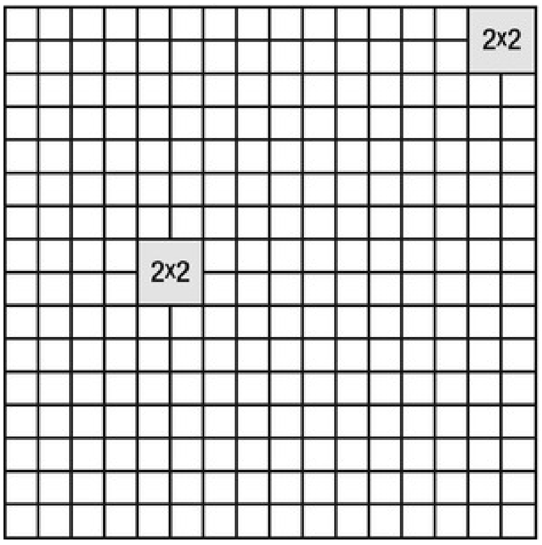


Figure 6-12 . A 16 × 16 matrix is divided into tiles of 2 × 2 each. Every four threads that belong in the same tile can share data between them

Within the GPU kernel, `idx.global` can be used in place of the standard `index < 2 >` we have seen earlier when performing operations on matrices. However, clever use of the local tile memory and local tile indices can reap significant performance benefits. To declare tile-specific memory that is shared across all threads in the same tile, the `tile_static` storage specifier can be applied to local variables inside the kernel. It is common to declare a shared memory location and have each thread in the tile initialize a small part of it:

```
parallel_for_each(tiledMatrix, [=](tiled_index < 2,2 > idx) restrict(amp) {
  tile_static int local[2][2]; //32 bytes shared between all threads in the tile
  local[idx.local[0]][idx.local[1]] = 42; //assign to this thread's location in the array
});
```

Clearly, procuring any benefits from memory shared with other threads in the same tile is only possible if all the threads can synchronize their access to the shared memory; i.e., they shouldn't attempt to access shared memory locations before their tile neighboring threads have initialized them. `tile_barrier` objects synchronize the execution of all the threads in a tile—they can proceed executing after calling `tile_barrier.wait` only after all the threads in the tile have called `tile_barrier.wait` as well (this is similar to the TPL's `Barrier` class). For example:

```
parallel_for_each(tiledMatrix, [](tiled_index < 2,2 > idx) restrict(amp) {
  tile_static int local[2][2]; //32 bytes shared between all threads in the tile
  local[idx.local[0]][idx.local[1]] = 42; //assign to this thread's location in the array
  idx.barrier.wait(); //idx.barrier is a tile_barrier instance
  //Now this thread can access "local" at other threads' indices!
});
```

It is now time to apply all this knowledge to a concrete example. We will revisit the matrix multiplication algorithm previously implemented without tiling, and introduce a tiling-based optimization into it. Let's assume that the matrix dimensions are divisible by 256—this allows us to work with 16 × 16 thread tiles. Matrix multiplication contains inherent blocking, which can be used to our advantage (in fact, one of the most common optimizations of extremely large matrix multiplication on the CPU is by using blocking to obtain better cache behavior). The primary observation boils down to the following. To find $C_{i,j}$ (the element at row *i* and column *j* in the result matrix), we have to find the scalar product between $A_{i,*}$ (the entire *i*-th row of the first matrix) and $B_{*,j}$ (the entire *j*-th row of the second matrix). However, this is equivalent to finding the scalar products of partial rows and partial columns and summing the results together. We can use this to translate our matrix multiplication algorithm to the tiled version:

```
void MatrixMultiply(int* A, int m, int w, int* B, int n, int* C) {
  array_view < const int,2 > avA(m, w, A);
  array_view < const int,2 > avB(w, n, B);
  array_view < int,2> avC(m, n, C);
  avC.discard_data();
  parallel_for_each(avC.extent.tile < 16,16 > (), [=](tiled_index < 16,16 > idx) restrict(amp) {
    int sum = 0;
    int localRow = idx.local[0], localCol = idx.local[1];
    for (int k = 0; k < w; k += 16) {
    tile_static int localA[16][16], localB[16][16];
    localA[localRow][localCol] = avA(idx.global[0], localCol + k);
    localB[localRow][localCol] = avB(localRow + k, idx.global[1]);
    idx.barrier.wait();
    for (int t = 0; t < 16; ++t) {
    sum + = localA[localRow][t]*localB[t][localCol];
    }
    idx.barrier.wait(); //to avoid having the next iteration overwrite the shared memory
    }
    avC[idx.global] = sum;
  });
}
```

The essence of the tiled optimization is that each thread in the tile (there are 256 threads, the tile is 16 × 16) initializes its own element in the 16 × 16 local copies of sub-blocks from the *A* and *B* input matrices (see Figure 6-13). Each thread in the tile needs only one row and one column of these sub-blocks, but all the threads together will access every row 16 times and every column 16 times, reducing significantly the number of main memory accesses.

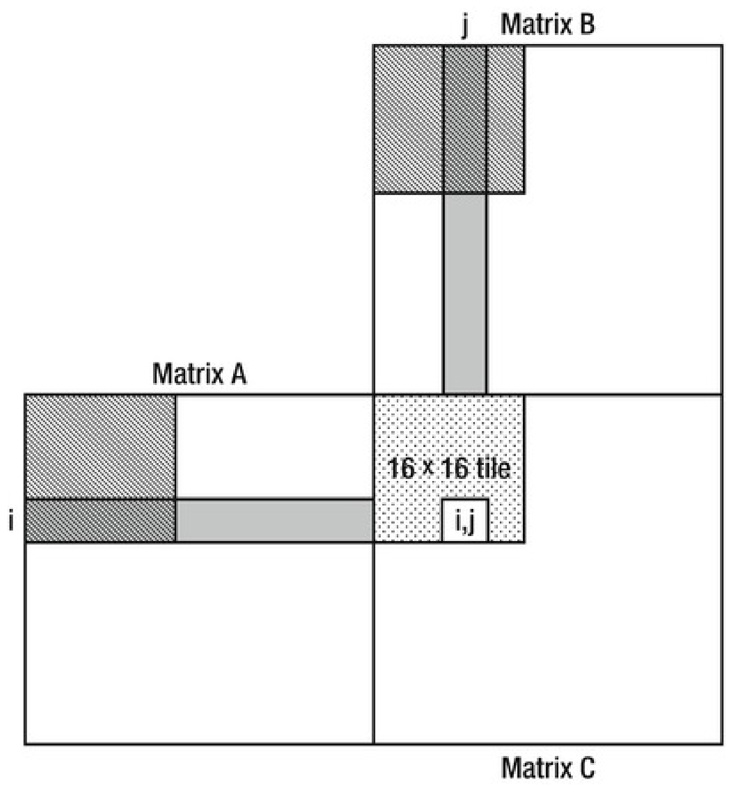


Figure 6-13 . To find the element (i,j) in the result matrix, the algorithm requires the entire i-th row of the first matrix and j-th column of the second matrix. When the threads in the 16 × 16 tile shown in the diagram execute and k = 0, the shaded areas in the first and second matrices will eventually be read into shared memory. The thread responsible for the (i,j)-th element in the result matrix will then have the partial scalar product of the first k elements from the i-th row with the first k elements from the j-th row

In this case, tiling is a worthwhile optimization. The tiled version of matrix multiplication executes significantly faster than the simple one, and takes 17 ms on average to complete (using the same 1024 × 1024 matrices). This concludes a *430-fold speed increase* compared to the CPU version!

Before we part with C++ AMP, it's worthwhile to mention the development tools (Visual Studio) available to C++ AMP developers. Visual Studio 2012 features a GPU debugger, which you can use to place breakpoints in GPU kernels, inspect simulated call stacks, read and modify local variables (some accelerators support GPU debugging; with others, Visual Studio uses a software emulator), and a profiler that can be used to gauge what your application has gained from using GPU parallelization. For more information on the Visual Studio GPU debugging experience, consult the MSDN article "Walkthrough: Debugging a C++ AMP Application", at `http://msdn.microsoft.com/en-us/library/hh368280(v=VS.110).aspx`.

.NET ALTERNATIVES FOR GPGPU COMPUTING

Although so far this entire section has dealt exclusively with C++, there are several options for harnessing the power of the GPU from managed applications. One option is to use managed-native interoperability (discussed in Chapter 8), deferring to a native C++ component to implement the GPU kernels. This is a reasonable option if you like C++ AMP, or have a reusable C++ AMP component that is used in managed applications as well as native ones.

Another option is to use a library that works with the GPU directly from managed code. There are several such libraries available, for example GPU.NET and CUDAfy.NET (both commercial offerings). Here is an example from GPU.NET GitHub repository, demonstrating a scalar product of two vectors:

```
[Kernel]
public static void MultiplyAddGpu(double[] a, double[] b, double[] c) {
 int ThreadId = BlockDimension.X * BlockIndex.X + ThreadIndex.X;
 int TotalThreads = BlockDimension.X * GridDimension.X;
 for (int ElementIdx = ThreadId; ElementIdx < a.Length; ElementIdx += TotalThreads) {
 c[ElementIdx] = a[ElementIdx] * b[ElementIdx];
}
}
```

In the authors' opinion, language extensions (the C++ AMP approach) are more effective and easier to learn than attempts to bridge the gap purely at the library level, or by introducing significant IL rewriting.

This section hardly scratches the surface of the possibilities offered by C++ AMP. We have only taken a look at some of the APIs and parallelized an algorithm or two. If you are interested in more details about C++ AMP, we strongly recommend Kate Gregory's and Ade Miller's book, "C++ AMP: Accelerated Massive Parallelism with Microsoft Visual C++" (Microsoft Press, 2012).