Deficiencies of .NET CLR JIT compilers

Another reason to use a GPU

Dr. Daniel Egloff
Managing Director QuantAlea
Microsoft MVP
June 9, 2016

Who are We

Software and solution provider for high performance and GPU computing, subsidiary of InCube

Technology driven financial services company

Managed Code Efficiency

QuantAlea

- Can managed code be as efficient as native code?
- In some cases optimized JIT code is as fast as native!
- For many cases optimized JIT code is significantly slower
  - 40% to 400% performance degradation

- JIT optimization tradeoff
  - Balance increased JIT compilation time against improved execution efficiency
  - Additional safety guards also in release mode
    - Range checks for array access
  - SSE vector instructions

Side Note

QuantAlea

- Inspecting generated MSIL code is not enough
- Need to look **at assembly code** generated by JIT compiler in **release mode**

- How to:
  - In project settings set **Generate debug info** to **pdb-only**
  - Go to **Tools → Options → Debugging → General**
    - Uncheck **Suppress JIT optimization on module load**
    - Uncheck **Enable Just My Code**
  - Compile in release mode and set breakpoint code
  - Run with debugger
  - Once debugger stopped right click and select **Go To Disassembly**

Example 1 – Simple Loop

QuantAlea

- Write to array in a loop

```
for (var i = 0; i < a.Length; ++i)
{
    a[i] = i;
}
```

Example 1 – Simple Loop

QuantAlea

```
for (var i = 0; i < a.Length; ++i)
00007FFF309244F0  xor          eax,eax
00007FFF309244F2  mov          edx,dword ptr [rcx+8]
00007FFF309244F5  test         edx,edx
00007FFF309244F7  jle          00007FFF30924507
{
    a[i] = i;
00007FFF309244F9  movsxd       r8,eax
00007FFF309244FC  mov          dword ptr [rcx+r8*4+10h],eax
for (var i = 0; i < a.Length; ++i)
00007FFF30924501  inc          eax
00007FFF30924503  cmp          edx,eax
00007FFF30924505  jg           00007FFF309244F9
00007FFF30924507  ret
```

Optimized

Example 2 – Nested Loops

QuantAlea

- Read from array and write to array in nested loops

```
for (var i = 0; i < lda; ++i)
{
    for (var j = 0; j < lda; ++j)
    {
        a[i*lda + j] = b[j*lda + i];
    }
}
```

Example 2 – Nested Loops

QuantAlea

```
. . .
for (var j = 0; j < lda; ++j)
00007FFF30944521  test        r8d,r8d
00007FFF30944524  jle         00007FFF30944560
00007FFF30944526  mov         r10d,dword ptr [rdx+8]
00007FFF3094452A  mov         r11d,eax
00007FFF3094452D  imul        r11d,r8d
00007FFF30944531  mov         esi,dword ptr [rcx+8]
                  {
                      a[i*lda + j] = b[j*lda + i];
00007FFF30944534  mov         edi,r9d
00007FFF30944537  imul        edi,r8d
00007FFF3094453B  add         edi,eax
00007FFF3094453D  cmp         edi,r10d
00007FFF30944540  jae         00007FFF3094456F

. . .
```

Range check not optimized

Example 3 – Floyd Warshall

QuantAlea

- Solving all shortest path problem – with 2d arrays

```
public static void FloydWarshall(int[,] d, int[,] p)
{
    var n = d.GetLength(0);
    for (var u = 0; u < n; u++) {
        for (var v1 = 0; v1 < n; v1++) {
            for (var v2 = 0; v2 < n; v2++) {
                var newPath = d[v1, u] + d[u, v2];
                var oldPath = d[v1, v2];
                if (oldPath > newPath) {
                    d[v1, v2] = newPath;
                    p[v1, v2] = p[u, v2];
                }
            }
        }
    }
}
```

Example 3 – Floyd Warshall

QuantAlea

- Solving all shortest path problem – flatten to improve performance

```
public static void FloydWarshall(int n, int[] d, int[] p)
{
    for (var u = 0; u < n; u++) {
        for (var v1 = 0; v1 < n; v1++) {
            for (var v2 = 0; v2 < n; v2++) {
                var newPath = d[v1 * n + u] + d[u * n + v2];
                var oldPath = d[v1 * n + v2];
                if (oldPath > newPath) {
                    d[v1 * n + v2] = newPath;
                    p[v1 * n + v2] = p[u * n + v2];
                }
            }
        }
    }
}
```

Floyd Warshall Performance

QuantAlea

- Comparing against C++ native
- Small size problem 1000 nodes 150000 edges
- I7-4770K @ 3.5Gz
  - C# 2.3 s
  - Microsoft C++ 1.7
  - C# roughly 35% slower

Moving to GPU

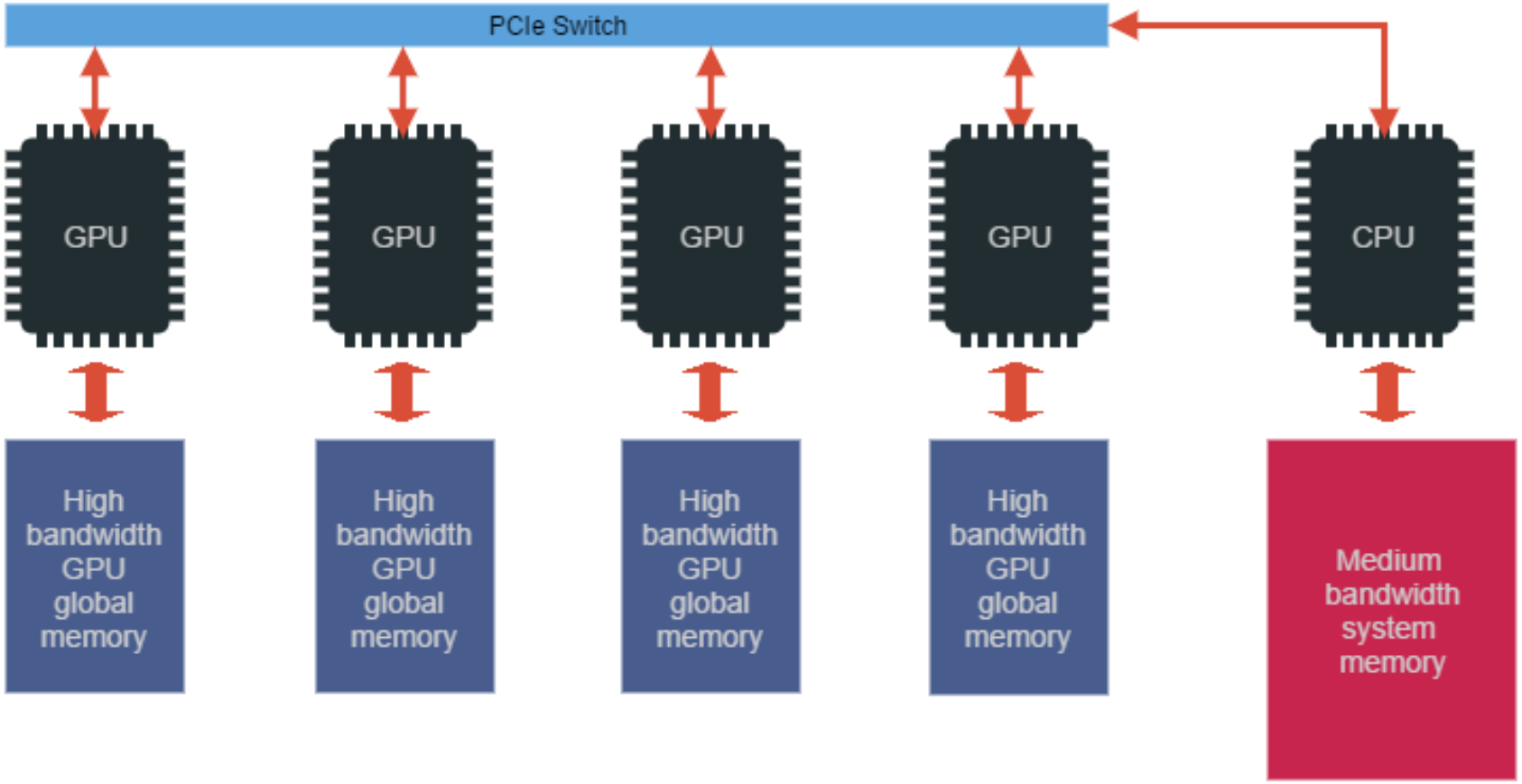
QuantAlea

- Often CRL JIT compilers cannot properly optimize numerical codes
  - Linear algebra
  - Search algorithms
  - Numerical optimization
- Moving to unmanaged C++ and using PInvoke leads to unwanted maintenance effort and language diversity

- Many numerical algorithms show some degree of parallelism
- Move to GPU to get hyper speedup!

GPU is a Coprocessor

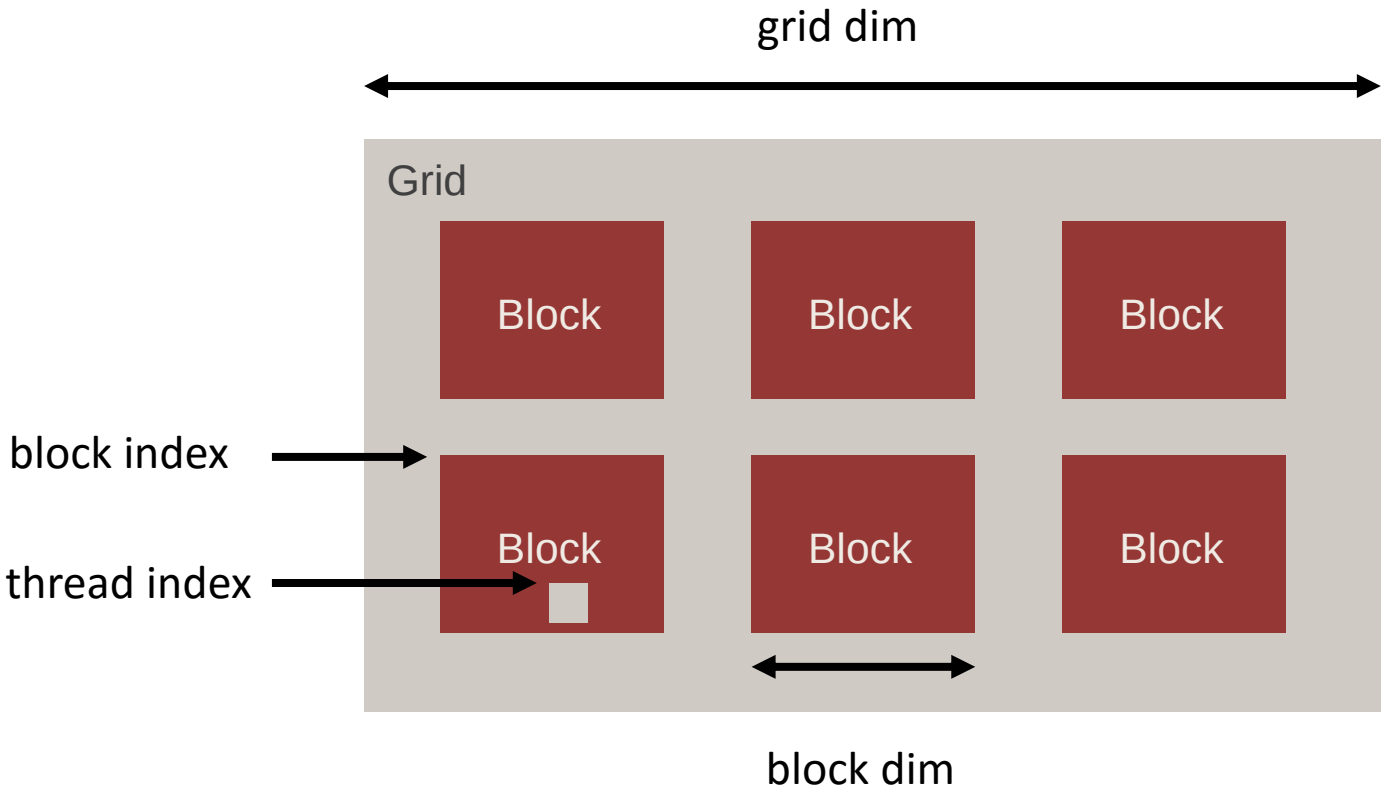
QuantAlea



Basic CUDA Concepts

QuantAlea

- Kernel
- Grid and thread block



FW GPU – Simple Implementation

QuantAlea

- GPU kernel with .NET types

```
public static void KernelSingleStage(int u, int[,] d, int[,] p)
{
    var n = d.GetLength(0);
    var v1 = blockDim.y * blockIdx.y + threadIdx.y;
    var v2 = blockDim.x * blockIdx.x + threadIdx.x;

    if (v1 < n && v2 < n)  {
        var newPath = d[v1, u] + d[u, v2];
        var oldPath = d[v1, v2];
        if (oldPath > newPath) {
            d[v1, v2] = newPath;
            p[v1, v2] = p[u, v2];
        }
    }
}
```

FW GPU – Simple Implementation

QuantAlea

- GPU kernel launching

```
[GpuManaged]
public static void Run(Gpu gpu, int[,] d, int[,] p)
{
    var n = d.GetLength(0);
    var gridDim =
        new dim3((n - 1)/BlockWidth + 1, (n - 1)/BlockWidth + 1, 1);
    var blockDim = new dim3(BlockWidth, BlockWidth, 1);
    var lp = new LaunchParam(gridDim, blockDim);
    for (var u = 0; u < n; u++) {
        gpu.Launch(KernelSingleStage, lp, u, d, p);
    }
}
```

FW GPU – Simple Implementation

QuantAlea

- Comparing against CUDA C++ native
- Small size problem 1000 nodes 150000 edges
- Tesla 40c with I7-4770K @ 3.5Gz
  - C# 2.3 s
  - GPU C# 6.2 s
  - **GPU C# 2.7 times slower than C# CPU**
  - **GPU C# 85 times slower than native CUDA C++**

- Issues
  - Memory must be well aligned, i.e. using pitched memory and device pointers
  - Only copy minimal amount of data, i.e. explicit memory management

FW GPU – Optimization I

QuantAlea

- GPU kernel with GPU optimized types

```
public static void KernelSingleStage(int u, int n, int pitch,
    deviceptr<int> d, deviceptr<int> p)
{
    var v1 = blockDim.y * blockIdx.y + threadIdx.y;
    var v2 = blockDim.x * blockIdx.x + threadIdx.x;

    if (v1 < n && v2 < n) {
        var newPath = d[v1*pitch + u] + d[u*pitch + v2];
        var oldPath = d[v1*pitch + v2];
        if (oldPath > newPath) {
            d[v1*pitch + v2] = newPath;
            p[v1*pitch + v2] = p[u*pitch + v2];
        }
    }
}
```

FW GPU – Optimization I

QuantAlea

- GPU kernel launching with explicit pitched memory management

```
public static Tuple<int[,], int[,]> Run(Gpu gpu, int[,] dh, int[,] ph)
{
    using (var d = gpu.AllocateDevice(dh))
    using (var p = gpu.AllocateDevice(ph)) {
        var n = dh.GetLength(0);
        var gridDim =
            new dim3((n - 1)/BlockWidth + 1, (n - 1)/BlockWidth + 1, 1);
        var blockDim = new dim3(BlockWidth, BlockWidth, 1);
        var lp = new LaunchParam(gridDim, blockDim);
        var pitch = (int)d.PitchInElements;
        for (var u = 0; u < n; u++) {
            gpu.Launch(KernelSingleStage, lp, u, n, pitch, d.Ptr, p.Ptr);
        }

        var dRes = Gpu.Copy2DToHost(d);
        var pRes = Gpu.Copy2DToHost(p);
        return new Tuple<int[,], int[,]>(dRes, pRes);
}}
```

FW GPU – Optimization I

QuantAlea

- Comparing against CUDA C++ native
- Small size problem 1000 nodes 150000 edges
- Tesla 40c with I7-4770K @ 3.5Gz
  - GPU C# with pitched memory explicitly managed 0.075 s
  - **GPU C# 30 times faster**
  - CUDA C++ 0.073 s

- Issues
  - Overall 2.7% performance degradation against CUDA C++
  - Inspecting with profiler: kernel execution times are identical
  - Launching GPU kernels from .NET has a marshalling overhead
  - Arrays must be pinned before sending to GPU

FW GPU – Optimization II

QuantAlea

- Change to blocked Floyd-Warshall algorithm
- Improve cache efficiency by using shared memory
- Minimize shared memory utilization
- Reduce instruction count
- Use less expensive instructions
- Price
  - More code
  - More complex algorithm
  - Requires detailed understanding of GPUs and parallel algorithms
- Benefit
  - Factor 2 to 5 speedup

FW GPU – Optimization II

QuantAlea

- Tesla 40c with I7-4770K @ 3.5Gz
- Small size problem 1000 nodes 150000 edges
  - GPU C# with pitched memory explicitly managed 0.037 s
  - CUDA C++ 0.034 s
  - GPU C# version **60 times faster** than initial C# version
- Larger problem 10000 nodes 2000000 edges
  - C# **2082 s approx 0.5 h**
  - GPU C# and CUDA C++ 46.4 s (Opt I) **16.5 s** (Opt II)
  - GPU C# version **125 times faster** than initial C# version
  - Overhead as compared to native CUDA C++ implementation immaterial

Wait a sec, I am lost!

QuantAlea

Does it need to be so complicated?

Fortunately not!

Use parallel for and parallel aggregate

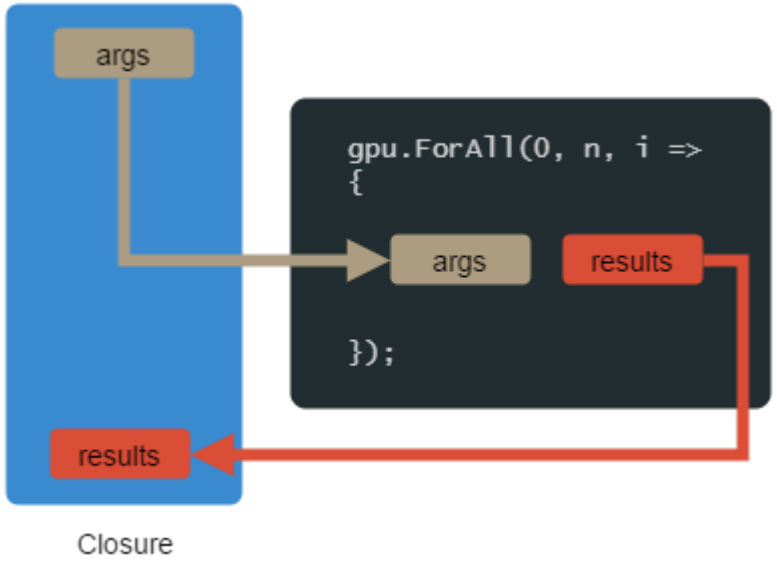
Parallel-For

QuantAlea

- Independent operation for each
  - element of a collection
  - index of an ordered range

```
void Gpu.For(int start, int end, Action<int> op);
```

- Arguments and the result are captured in a closure and passed to the parallel-for body



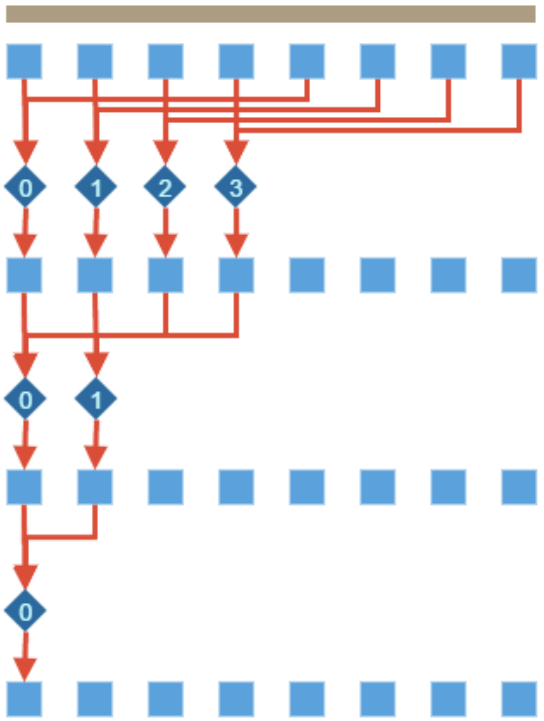
Parallel Aggregate

QuantAlea

- Reduce collection of elements with an associative binary operator to a single value.

```
T Gpu.Aggregate<T>(T[] elements, Func<T, T, T> op);
```

- Convenience overloads for
  - Sum
  - Average
- Highly efficient implementation
  - Comparable to NVIDIA CUB



GPU Computing Made Simple

QuantAlea

```
int n = 1000;
var rng = new System.Random();
var x = Enumerable.Range(0, n).Select(_ =>
    rng.NextDouble()).ToArray();
var y = Enumerable.Range(0, n).Select(_ =>
    rng.NextDouble()).ToArray();
var res = new double[n];

Parallel.For(0, n, i => res[i] = x[i] + y[i]);

var expected = x.Zip(y, (xi, yi) => xi + yi);

Assert.That(res, Is.EqualTo(expected));

var sum = x.AsParallel().Aggregate((xi, yi) => xi + yi);

Assert.That(sum, Is.EqualTo(x.Sum()).Within(1e-12));
```

GPU Computing Made Simple

QuantAlea

```
int n = 1000;
var rng = new System.Random();
var x = Enumerable.Range(0, n).Select(_ =>
    rng.NextDouble()).ToArray();
var y = Enumerable.Range(0, n).Select(_ =>
    rng.NextDouble()).ToArray();
var res = new double[n];

Gpu.Default.For(0, n, i => res[i] = x[i] + y[i]);

var expected = x.Zip(y, (xi, yi) => xi + yi);

Assert.That(res, Is.EqualTo(expected));

var sum = Gpu.Default.Aggregate(x, (xi, yi) => xi + yi);

Assert.That(sum, Is.EqualTo(x.Sum()).Within(1e-12));
```

Same delegate

Pointers

QuantAlea

- Alea GPU V3
  - http://www.aleagpu.com/release
  - Samples in sample gallery http://www.aleagpu.com/release/3_0_0-beta10/doc/gallery.html
  - Alea GPU 2.x community license also works for Alea GPU V3
- Alea GPU V3 beta NuGet packages
  - http://beta.aleagpu.com/nuget
- Alea TK
  - New open source deep learning stack based on Alea GPU
  - To be released around September 2016

Contact details

Dr. Daniel Egloff
Phone: +41 79 430 03 61
daniel.egloff@quantalea.net
daniel.egloff@incubegroup.com