Runtime-based approach to a parallel programming model design

Andrei Varanovich
Agenda

- Quick look into data parallel approach
- Massively parallel semantics: challenges and opportunities
- Case study: CLR-based framework and programming model for massively-parallel operations
Q: Is **SUM** operation parallel or sequential?

A: Parallel

6 \(\oplus\) 3 = 9

Bit Twiddling Hacks
http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetParallel

SSE 2 Optimizations
http://bmagic.sourceforge.net/bmsse2opt.html
Key concept: type and operation
Y = αX + Y

Where α – scalar; X, Y - vectors
C++

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

Haskell

```
saxpy alpha xs ys =
    zipWith (\x y -> alpha*x + y) xs ys
```

LISP

```
Why should I care?
```
Bring SAXPY to parallel

- Focus on data dependencies: execution order does not matter
- Focus on what we want to achieve rather than how

Apply operation $Y$ on elements of type $A$ and $B$ in parallel
Data parallel future today

Intel I7

NVidia Tesla C2050

8 Computing Cores

448 Computing Cores
We need the way to express massively parallel semantics
Hierarchical memory
Restricted address space
Data uploading

Flat memory
Virtual memory
Direct access
__global__ void Saxpy(float a, float* X, float* Y)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    Y[i] = a * X[i] + Y[i];
}

cudaMemcpy(X, hX, cudaMemcpyHostToDevice);
cudaMemcpy(Y, hY, cudaMemcpyHostToDevice);
Saxpy<<<256, (N + 255) / 256>>>(a, hX, hY);
cudaMemcpy(hY, Y, cudaMemcpyDeviceToHost);
Compiler-centric approach

- Compiler oriented
- Huge gap between high-level programming language constructs and low-level details
type Vector = Array DIM1 Float

saxpy :: TP1.Expr Float -> Vector -> Vector -> Vector

saxpy alpha xs ys
= TP1.run $ do
  xs' <- use xs
  ys' <- use ys
  TP1.zipWith (\x y -> alpha*x + y) xs' ys'

TPx e — expression evaluated on the target architecture
TPx.run — compile & execute embedded code
How to deal with TPx internals?
Runtime-based approach

- Compiler
- Target assembler
- Execution
- Runtime

- Runtime-oriented
- Runtime cares about: low-level implementation details
Runtime: main attributes

- Manages an execution of the application
- Manages the application’s memory
- Offers high-level representation of underlying hardware/OS features
Case study

Conflux

A framework for massively parallel heterogeneous environments
Hi, with the advent of nVidia's CUDA etc, is there going to be support for programming against a GPU in .Net 4.0? There seems to be a large focus on parallelism as regards multicore CPUs, but haven't seen anything about GPUs.

Cheers.
GPGPU programming with C# and F#

People often ask me about GPGPU (General Purpose GPU) programming with F#.

One project you might like to take a look at is Brahma, which lets you compile C# expression trees to the GPU. When combined with the F# Power Pack support for converting F# quotations to LINQ expression trees, this should give a viable technique for using GPUs for computation.

Another route is to use Microsoft Accelerator, an incubation project from Microsoft Research. New functionality includes...
Utilization of the existing .NET ecosystem (Languages, IDE, debuggers, etc.)

Targeting all programming languages for .NET Framework

Native support (no 3rd-party pre- post-processing)
What gives us a dynamic runtime

- Automatic multi-level memory management
- Dynamic workload balancing
- Runtime optimizations
- Interoperability with existing libraries
How does it work?

- Front end decompiles C#.
- AST transformer inlines calls, destructures classes and arrays.
- Back end generates PTX (NVIDIA GPU assembler) and/or multicore IL.
- Interop binds to nvcuda driver that is capable of executing GPU assembler.
```java
public abstract class MatMulKernel : Kernel<float[], float[], float[]>
{
    protected override void Initialize(IGridConfig gridCfg, float[] a, float[] b)
    {
        if (gridCfg.GridDim != null && gridCfg.BlockDim != null) return;

        const int defaultBlockSize = 16;
        var blockDim = gridCfg.BlockDim ?? new dim3(Min(_b.Width(), defaultBlockSize), Min(_a.Height(), defaultBlockSize));
        gridCfg.SetDims(gridDim, blockDim);
    }

    protected float[,] _a;
    protected float[,] _b;
    [Result] protected float[,] _c;

    protected override void RunKernel()
    {
        var row = BlockIdx.Y * blockDim.Y + ThreadIdx.Y;
        var col = BlockIdx.X * blockDim.X + ThreadIdx.X;
        // this is necessary in case when matrix dims ain't multiples of block dims
        if (_a.Height() <= row || _b.Width() <= col) return;

        var c_value = 0f;
        for (var dim = 0; dim < _a.Width(); ++dim)
        {
            c_value += _a[row, dim] * _b[dim, col];
        }

        _c[row, col] = c_value;
    }
}
```
public interface IKernel
{
    void Initialize(IGridConfig gridCfg, params Object[] args);
    void RunKernel();
    Object FetchResult();
}
var cuda = new CudaConfig{ISA = ISA.SM_13};
var compiledKernel = cuda.Configure<MatMulKernel_Fast>();
var result = compiledKernel.Execute(..., ...);
Behind the scenes

- CPU -> map kernels to CPU cores
- GPU -> map kernels one-to-one to CUDA kernels
- Statically typed
- No extra constructs and/or language extensions
Data-parallel approach has proven its benefits (e.g. MapReduce)

Runtime gives advantages
- Abstract data types instead pre-defined “parallel arrays” (e.g. Data-parallel Haskell)
- Extensibility (unlimited number of underlying hardware architectures)
- Level of abstraction together with massively parallel semantics
Questions

dotnetby@gmail.com

varanovich@koblenz-landau.de (as soon as I get access 😊 )
References

Thank you!