How to design a language integrated compiler with LLVM

GTC 2014
San José, California

March 27, 2014

Xiang Zhang (xiang.zhang@quantalea.net)
Aaron Brewbaker (aaron.Brewbaker@quantalea.net)
About Us

Software development
- Quantitative finance and risk management
- Derivative pricing and modeling
- Numerical computing
- High performance computing (clusters, grid, GPUs)
- Software engineering (C++, F#, Scala, …)
- Early adopter of GPUs
- Creator of Alea.cuBase

Consulting and advisory
- Risk management, quantitative finance, reporting, regulations
- Banks, insurance firms, asset management firms, energy and commodity trading firms
- Interdisciplinary team, with expertise in methodology, implementation, IT architecture
- Using QuantAlea technology in projects
Topics

- Motivation

- Technologies
  - NVVM compiler
  - Design of Alea.cuBase
  - Parallel primitives library
Why is CUDA C/C++ popular?

- Provides a good abstraction for GPU programming (common purpose, low level memory control, scalable)
- Heterogeneous programming by extending the C language
- Easy to learn because C is well known in comparison to a special shader language
- Reuses existing C libraries for CPU programming
- Integrated into sophisticated IDEs (Visual Studio, Eclipse, Nsight)
- Cooperates with other languages because many languages can call C libraries
- Provides PTX ISA and JIT driver to adapt to different and future hardware
- Plenty of tutorials, books and professional programmers
Productivity is still poor in comparison to modern languages and tools such as C#, F#, Java, Scala, etc. which are built on top of modern runtime environments with thousands of libraries

No scripting support as with Python, Ruby, Matlab, R, etc.

A lot of boiler plate code for integration, memory management, etc.

Error prone - null pointers, implicit conversions, etc.

Rigid in the sense that it allows only statically compiled and linked applications

Not easy to setup development environment

Not easy to deploy
Kernel language as DSL
- SIMD primitives: threadIdx, blockDim, etc.
- Low level memories: shared, constant, cache, vector data accessing, etc.
- Special functions: thread barrier, memory barrier, atomic, etc.

Internal DSL
- Integrated into existing language
- Same syntax, flat learning curve
- Gain productivity via existing language tools (IDE, grammar checking, debugger), libraries, scripting, etc.

Runtime compiling
- Compiling and linking features provided by library API
- Dynamic kernel generation on-demand
- Flexibility in targeting different architectures and device capabilities

Managing runtime CUDA objects
- Device memories and symbols
- Kernel loading and launching
- Stream
Topics

- Motivation

Technologies

- NVVM compiler
- Design of Alea.cuBase
- Parallel primitives library
LLVM compiler

High level languages: C/C++, C#, F#, Python, Ruby, etc.

- Parsing
- Abstract Syntax Tree (AST)
- Translating

Intermediate Representation (LLVM IR byte-code)

- Optimization and Machine-code Emitting

Target Machine-code: x86, ARM, PowerPC, PTX

Front-end

Back-end
NVVM compiler

NVVM IR Module
(Subset of LLVM IR with GPU programming model)

NVVM Compiler
(A C library provides compiling and linking API)

PTX code
OpenCL workflow

- All done at runtime – could cache the OpenCL binary
- Different vendors provide their own implementation
## CUDA vs. OpenCL

<table>
<thead>
<tr>
<th></th>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel Language</td>
<td>CUDA C/C++</td>
<td>OpenCL C</td>
</tr>
<tr>
<td>IR</td>
<td>LLVM/NVVM</td>
<td>Implemented by vendors</td>
</tr>
<tr>
<td>ISA</td>
<td>PTX</td>
<td>Implemented by vendor</td>
</tr>
<tr>
<td>Machine code</td>
<td>SASS</td>
<td></td>
</tr>
<tr>
<td>Frontend Compiler</td>
<td>NVCC Compiler</td>
<td></td>
</tr>
<tr>
<td>Backend Compiler</td>
<td>NVVM Compiler</td>
<td></td>
</tr>
<tr>
<td>JIT Compiler</td>
<td>CUDA Driver</td>
<td></td>
</tr>
</tbody>
</table>
Modern languages

- Diversified programming paradigm
  - Static language: C#, F#, Java, etc.
  - Dynamic language: Python, Ruby, etc.
  - Strongly typed: F#, etc.
  - Functional: F#, Scala, etc.

- Meta-programming capable
  - Reflection: C#, F#, Scala, etc.
  - AST: F# quotation, Scala, Python, Ruby, etc.
  - Type Provider: F#

- Gain flexibility by virtual machine technology: .NET IL, Java byte-code, etc.
- Plenty of third-party libraries
- Easy development and deployment
  - Sophisticated IDEs and tools
  - Well-designed package management: NuGet, etc.
- Interop capabilities with C library which make it possible to call NVVM library and CUDA Driver API
## Existing products

- **CUDAfy**: Uses .NET CLR reflection to “CUDAfy” a C# class, translate it into CUDA C/C++ code, then use nvcc to compile.
- **NumbaPro**: Uses Python metaprogramming feature and NVVM technology.
- **Alea.cuBase**: Uses F# quotation to get AST and NVVM technology.
Topics

- Motivation
- Technologies
  - NVVM compiler
  - Design of Alea.cuBase
  - Parallel primitives library
NVVM has three parts: NVVM IR specification, NVVM compiler API, LibDevice.

Detailed document can be found at http://docs.nvidia.com
```c
char *generatePTX(const char *ll, size_t size, const char *filename)
{
    nvvmResult result;
    nvvmProgram program;
    size_t PTXSize;
    char *PTX = NULL;

    result = nvvmCreateProgram(&program);
    if (result != NVVM_SUCCESS) {

    }

    result = nvvmAddModuleToProgram(program, ll, size, filename);
    if (result != NVVM_SUCCESS) {

    }

    result = nvvmCompileProgram(program, 0, NULL);
    if (result != NVVM_SUCCESS) {

    }

    result = nvvmGetCompiledResultSize(program, &PTXSize);
    if (result != NVVM_SUCCESS) {

    }

    PTX = (char*)malloc(PTXSize);
    result = nvvmGetCompiledResult(program, PTX);
    if (result != NVVM_SUCCESS) {

    }

    result = nvvmDestroyProgram(&program);
    if (result != NVVM_SUCCESS) {

    }

    return PTX;
}
```
CUDA Driver API

```c
#if BUILD_64_BIT
    const char *filename = "simple-gpu64.ll";
#else
    const char *filename = "simple-gpu.ll";
#endif
char *ll = loadProgramSource(filename, &size);
fprintf(stdout, "NVVM IR ll file loaded\n");

// Use libnvvm to generate PTX
ptx = generatePTX(ll, size, filename);
fprintf(stdout, "PTX generated:\n");
fprintf(stdout, "%s\n", ptx);

// Initialize the device and get a handle to the kernel
checkCudaErrors(initCUDA(&hContext, &hDevice, &hModule, &hKernel, ptx));

// Allocate memory on host and device
if ((h_data = (int *)malloc(memSize)) == NULL) {
    checkCudaErrors(cuMemAlloc(&d_data, memSize));
}

// Launch the kernel
checkCudaErrors(cuLaunchKernel(hKernel, nBlocks, 1, 1, nThreads, 1, 1, 0, NULL, params, NULL));
fprintf(stdout, "CUDA kernel launched\n");

// Copy the result back to the host
checkCudaErrors(cuMemcpyDtoH(h_data, d_data, memSize));
```
NVVM IR

- Subset of LLVM IR: supports most LLVM IR constructs and ignores meaningless constructs within GPU programming
- Organized global variables and functions within NVVM IR module
- Supports linking different NVVM IR modules into one PTX module
- Pre-defines several memory spaces to decorate pointers: shared, constant, global, or generic
- Uses virtual register (i.e. “LLVM Static Single Assignment”), leave the optimization job to NVVM compiler
- Provides built-in intrinsic functions for GPU programming: threads dimensions and location, memory and thread barrier.
- Supports inline PTX assembly language
- Provides pre-created NVVM IR module called “LibDevice” as kernel’s runtime library, which includes math functions, special SIMD functions, etc.. To use it, simply link it with your NVVM IR module
- Uses LLVM metadata to tag additional information: tagging kernel functions, adding source level debugging information, etc.
- Could be created programmatically by using LLVM API
target datalayout = "e-p:32:32-i1:8:8-i8:8:8-i16:16:16-..."

define void @$k0"(i32* %p0, i32* %p1) {
  entry:
    %0 = alloca i32*, align 4
    %1 = alloca i32*, align 4
    store i32* %p0, i32** %1, align 4
    store i32* %p1, i32** %0, align 4
    %2 = call i32 @llvm.nvvm.read.ptx.sreg.tid.x()
    %3 = sext i32 %2 to i64
    %4 = load i32** %1, align 4
    %5 = getelementptr inbounds i32* %4, i64 %3
    %6 = sext i32 %2 to i64
    %7 = load i32** %0, align 4
    %8 = getelementptr inbounds i32* %7, i64 %6
    %9 = load i32* %5, align 4
    store i32 %9, i32* %8, align 4
  ret void
}

declare i32 @llvm.nvvm.read.ptx.sreg.tid.x() nounwind readnone

!nvvm.annotations = !{!0}

!0 = metadata ![void (i32*, i32*)] @$k0", metadata !"kernel", i32 1}
Topics

- Motivation
- Technologies
  - NVVM compiler
  - Design of Alea.cuBase
  - Parallel primitives library
Alea.cuBase provides ...

- A complete solution to develop CUDA accelerated applications within the .NET ecosystem
- CUDA kernel authoring with the F# language
- Support of nearly every aspect of CUDA programming
- API to compile and link CUDA kernels
- Wrapping of CUDA driver to launch kernel in runtime
- Interoperability with various .NET libraries for productivity
- Good tooling support from Microsoft Visual Studio, Intellisense, Nsight, . . .
## We choose F#

<table>
<thead>
<tr>
<th>Rank</th>
<th>Change</th>
<th>Programming Language</th>
<th>Ratings</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>↑</td>
<td>F#</td>
<td>1.216%</td>
<td>+1.14%</td>
</tr>
<tr>
<td>13</td>
<td>↓</td>
<td>Perl</td>
<td>1.149%</td>
<td>-0.81%</td>
</tr>
<tr>
<td>14</td>
<td>↓</td>
<td>Ruby</td>
<td>0.974%</td>
<td>-1.18%</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>Delphi/Object Pascal</td>
<td>0.881%</td>
<td>-0.01%</td>
</tr>
</tbody>
</table>
Why we choose and love F#

- Modern functional language which supports multiple programming paradigms
- Static and strongly typed language with type inference technology
- Extendable
  - Reflection
  - Computation expression (aka monad or workflow)
  - Built-in code quotation feature to access code AST
- Reusable from other .NET languages such as C#, Visual Basic, etc.
- Open sourced and supported on Windows, Linux and Mac OS.
- Unique Type Provider technology, easily adapted to various real world data formats
- Strong tooling such as Visual Studio, Nsight, NuGet, etc.
- Scripting capability, can be integrated into Excel
- Easy setup of development environment
- Easy application deployment by .NET assembly technology
- Large collection of high quality third-party libraries
Define resources with template

- Template is created by “cuda” computation expression
- Device resources such as global variables and kernel/device functions are defined inside template
- A special resource “Entry” can be returned which provides a host entry point routine to manage runtime CUDA objects and launch the kernels.
- Template containing entry point can be loaded into device and executed.

```ml
let template = cuda {
  let! constData = Compiler.DefineConstantArray<int>(10)

  let! kernel =<@	
  fun (output:deviceptr<int>) (input:deviceptr<int>) ->

  ... @>

  |> Compiler.DefineKernel

  return Entry(fun program ->
    let worker = program.Worker
    let kernel = program.Apply kernel
    let run () =
      use output = worker.Malloc(...) 
      use input = worker.Malloc(...) 
      kernel.Launch LaunchParam(8, 128) output.Ptr input.Ptr 
      output.Gather()
    
    run ) }
```
Kernel authoring

- Kernels are authored via F# code quotations
- Provides pointer type and methods for low level memory access
- Provides special functions to define memories in different memory spaces such as shared memory and local memory
- Provides SIMD primitives and intrinsic functions such as threadIdx, blockDim, __syncthreads(), etc.
- Provides ways to use device resources defined in a template such as device functions, constant memories, etc.
- Supports LibDevice functions
- Supports enum, struct, class, tuple, F# record type
- Overrides most basic F# operators
let rec build (ctx:IRModuleBuildingContext) (expr:Expr) =
  match expr with
  | Let(var, valueExpr, followingExpr) ->
    let irValue = build ctx valueExpr
    bindVar var irValue
    build ctx followingExpr
  | Call(objectExprOpt, methodInfo, paramExprs) ->
    let irObject = objectExprOpt |> Option.map (build ctx)
    let irParams = paramExprs |> List.map (build ctx)
    buildCall methodInfo irObject irParams
    // in buildCall:
    // if a custom builder is found in the method attribute?
    // if there is a builder found in builder registry?
  | ...

Translate Quotation to IRValue
Compile time and runtime

return Entry(fun program ->
    let worker = program.Worker
    let kernel = program.Apply kernel

    let run (inputs:'T[]) (iters:int) =
        let n = inputs'.Length
        use inputs = worker.Malloc<'T>(n)
        use outputs = worker.Malloc<'T>(n)
        inputs.Scatter(inputs')
        let lp = LaunchParam(blocks, threads)
        use start = worker.CreateEvent()
        use stop = worker.CreateEvent()
        worker.Synchronize()
        start.Record()
        if profile then workerProfilerStart()
        for i = 0 to iters - 1 do
            kernel.Launch lp outputs.Ptr inputs.Ptr n
            if profile then workerProfilerStop()
        stop.Record()
        stop.Synchronize()

        let msecs = Event.ElapsedMilliseconds(start, stop)
        let msec = msecs / (float iters)
        let outputs = outputs.Gather()

        outputs, msec

    run )}
Compiling and linking

- Templates can be nested by referencing other templates (source level reusing)
- Multiple NVVM IR modules can be linked together (binary level reusing)
- The linked PTX module can be loaded on to CUDA device and then it returns a program object which contains the entry point function
Extendable architecture

- Provides various custom builder interfaces for different language constructs: function, type, methods, property, fields, runtime marshaling
- The basic kernel authoring constructs are built upon these custom builders
- Two ways to use custom building: attributes and/or builder registry

Utilities

Primitives  Intrinsic  overrides  libdevice  resource

Compiler Core  Runtime Engine

P/Invoke

CUDA Driver  NVVM Compiler  LLVM
Debugging

IR Module

IR debugging info
metadata

PTX module

Dwarf sections
Topics

- Motivation
- Technologies
- NVVM compiler
- Design of Alea.cuBase
- Parallel primitives library
A parallel kernel primitives library
Well optimized and good performance
Relies heavily upon C++ template technology
Utilize GPU hardware power by using inline PTX assembly
CUB library

- Aims to provide state-of-the-art-performance and reduce the maintenance burden of complex parallel code

- Relies heavily on C++ template meta-programming

- Kernel authoring via 3 levels of parallel primitives
  - Device-wide Primitives
    - Parallel sort, scan, etc.
  - Block-wide “collective” primitives
    - Cooperative I/O, sort, scan, etc.
    - Can use arbitrary thread block size
  - Warp-wide “collective” primitives
    - Cooperative warp-wide prefix scan
    - Architecture specialized

- Modular and adaptive
  - Design lends itself well to a functional language
CUB library

CUB makes extensive use of template meta-programming in order to fine-tune the algorithms for the best possible performance.
In Alea.cuBase, we can reproduce the sophisticated C++ meta-programming by using custom attribute builders

- Template specialization, loop-unrolling, and inline ptx are handled at kernel compile time.

- Allows for easier control and adaptability of performance-critical kernel specialization as compared to C++ template meta-programming.
Ensure solid performance by separating your parameters according to their temporal context.

```plaintext
<Record>
type StaticParam =
{
    // Template parameters
    BLOCK_THREADS : int
    ITEMS_PER_THREAD : int
    // Other kernel compile time constants
    SharedMemoryLength : int
}
<Record>
type InstanceParameters<'T> =
{
    mutable temp_storage : deviceptr<'T>
    mutable linear_tid : int
}
<Record>
static member Init(sp:StaticParam) =
  let ts = __shared__.Array<'T>(sp.SharedMemoryLength) |> __array_to_ptr
  { temp_storage = ts; linear_tid = threadIdx.x}
```
Performance Optimization Problem:
- Apply a cache load modifier to a thread load function

Performance Optimization Solution:
- Use an Alea.cuBase Custom Attribute Builder to implement inline-ptx during kernel compile time
First we define a function which gets passed the modifier, the IRModuleBuildingContext, and the IRValue representation of the target pointer.

```ocaml
let buildThreadLoad (modifier:string) (ctx:IRModuleBuildingContext) (irPointer:IRValue)
  let irPointerType = irPointer.Type
  let irPointeeType = irPointerType.Pointer.PointeeType

  let irPointerInt, ptrstr = ctx.CompileOptions.AddressSize | function
    | AddressSize.Address32 -> IRCommonInstructionBuilder.Instance.BuildPtrToInt(ctx)
    | AddressSize.Address64 -> IRCommonInstructionBuilder.Instance.BuildPtrToInt(ctx)

  in
    let irType = irPointerType.Ptr.PtrType in
    let irPointee = IRIntType(irPointeeType.Int.SizeType, 0) in
    let irValue = IRIntValue(irPointee, irPointerInt, ptrstr) in
    irValue
```

Match the address size to build the correct integer representation of the pointer.

Define a function which determines the pointee type.

```ocaml
let isUIntVector (bits:int) (dims:int) (irType:IRType) =
  irType.IsValStruct &&
  irType.Struct.FieldInfos.Length = dims &&
  Array.forall (fun (info:IRStructFieldInfo) ->
    isUIntInt bits info.FieldType) irType.Struct.FieldInfos
```
CUB library

Match the pointee type and construct the inline-px string

```c
let cmdstr, argstr = irPointeeType | function
  | irPointeeType when isU1ntVector 32 4 irPointeeType ->
    sprintf "%d.%s.4.u32 {%$0, $1, $2, $3}, [%$4];" modifier, sprintf "=%r,%r,%r,%r,%s" ptr
```

Lastly we define the attribute which will use the previous function to build and apply the ptx string

```c
[<AttributeUsage(AttributeTargets.Method, AllowMultiple = false)>]
type ThreadLoadAttribute(modifier:string) =
  inherit Attribute()

  interface ICustenCallBuilder with
    member this.Build(customCallInfo, irParams) =
      match irObject, modifier, ctx irPointer |
        | None, irPointee -> BuildThreadLoad modifier ctx irPointer |-> Some
        | _ -> None

let [<ThreadLoad("ca")>] inline ThreadLoad_CA (ptr:deviceptr<'T>): 'T = failwith "device only"
```
F# is a mature, first-class CUDA language which we are using in various client’s projects

- We rely heavily on NVVM technology and are therefore future proof
- We have debugging support based on LLVM meta-data
- Our F# CUDA language extensions seamlessly integrate with all F# technologies
- F# together with its CUDA extensions have proven to be an ideal platform for developing outstanding big data and machine learning applications
More Resources and Free Licenses

More resources
- https://www.quantalea.net/products/resources/
- http://blog.quantalea.net
- http://www.aleacubase.com/release/
- http://www.aleacubase.com/cudalab/

Machine learning with F# and CUDA
- https://github.com/SpiegelSoft/Vulpes.git

How to set up
- Fermi GPU device or better (compute capability >= 2.0)
- Windows with .NET 4 and F# 3.0
- CUDA 5.5 driver or newer
- Install Alea.cuBase, e.g. from NuGet gallery
- No need for CUDA toolkit or NVCC compiler
Thank you