Performance portability via Nim metaprogramming

Xiao-Yong Jin and James C. Osborn
Argonne Leadership Computing Facility, Argonne National Laboratory

A DOE ECP effort from Lattice QCD

August 23
DOE COE Performance Portability Meeting 2017
Denver, Colorado
“Complexity” seems to be a lot like “energy”: you can transfer it from the end user to one/some of the other players, but the total amount seems to remain pretty much constant for a given task.

—Ran, 5 Mar 2000
Outline

• Benchmark code in Nim
• Application domain: Lattice Gauge Theory
• Nim programming language
• onGPU
• Benchmark result
  • P100 + XEON
  • KNL
Benchmark

\[ X_{ij}^l \leftarrow X_{ij}^l + Y_{ik}^l \times Z_{kj}^l \]

3: proc test(vecLen, memLen: static[int]; N: int) =
4:   var
5:     x = newColorMatrixArray(vecLen, memLen, N)  Define complex 3×3 matrix field
6:     y = newColorMatrixArray(vecLen, memLen, N)
7:     z = newColorMatrixArray(vecLen, memLen, N)
32:   threads:
33:     x := 0
34:     y := 1
35:     z := 2
36:     timeit "CPU": x += y * z  Benchmark
37:     timeit "GPU5": onGpu(N, 32):
38:         x += y * z
39:     timeit "GPU6": onGpu(N, 64):
40:         x += y * z
41:     timeit "GPU7": onGpu(N, 128):
42:         x += y * z
43:     threads: timeit "CPU": x += y * z  Back to CPU threads
44:     threads: timeit "CPU": x += y * z  Rerun with different T/B

Define complex 3×3 matrix field
CPU threads
Set diagonal elements
Benchmark
Run statement block on GPU
Back to CPU threads
CPU threads

- Takes a block of code statements
- Wraps in a function with lexically scoped thread local objects
- References to variables outside the code block are managed by Nim
- Runs the function under omp parallel directive
- A custom iterator over the array indices takes care of actual data parallel operations

\[ x += y \times z \quad \Rightarrow \quad x[i] += (y \times z)[i] \]
AST based overloading for data parallel ops

1:  type \texttt{ArrayIndex*} = SomeInteger or ShortVectorIndex

2:  template \texttt{indexArray*}(x: ArrayObj, i: ArrayIndex): untyped =

3:    x.p[i]

4:  

5:  macro \texttt{indexArray*}(x: ArrayObj\{call\}, y: ArrayIndex): untyped =

6:    result = newCall(ident($x[0]))

7:  for i in 1..<x.len:

8:    let xi = x[i]

9:    result.add( quote do:

10:      indexArray(`xi`,`y`) )

11:  

12:  template `[]`*(x: ArrayObj, i: ArrayIndex): untyped =

13:    indexArray(x, i)

- When an ArrayObj is indexed, if the object is a function call, the indexing goes inside the call

\begin{align*}
x[i] & \ += (y \times z)[i] \\
& \rightarrow \quad x[i] \ += y[i] \times z[i]
\end{align*}
Lattice gauge theory

- Large 4D (5D) grid of small vectors/matrices with homogeneous stencil operations — large sparse linear algebra

\begin{align*}
q(x + \hat{v}) &
U_\mu(x + \hat{v}) &
q(x) &
U_\mu(x) &
q(x + \hat{u} + \hat{v}) &
U_\nu(x + \hat{u}) &
U_\nu(x + \hat{u} + \hat{v})
\end{align*}

\text{quark} \quad \text{gluon}
Nim

• Modern (since 2008) language
• “Efficient Expressive Elegant”
• Statically typed systems language (full access to low-level objects & code) with type inference
• Generates C or C++ code & compile with any compiler
• Integrated build system (no Makefile necessary): copy main program, modify, compile
• [https://nim-lang.org](https://nim-lang.org)
Nim—both low-level and high-level

- Low-level efficiency
  - Can manually manage memory instead of GC
  - Cross module inlining and constant unfolding
  - Whole program dead code elimination
- High-level wrappers & libraries
  - gmp, bignum, nimblas, linalg(LAPACK), ...
  - bindings to GTK2, the Windows API, the POSIX API, OpenGL, SDL, Cairo, Python, Lua, TCL, X11, libzip, PCRE, libcurl, mySQL, SQLite, ...
  - exportc to create static/dynamic libraries
- NimScript: shell-like scripting
  - Used in compiler for compile-time evaluation
  - Available to plug in to application and can interface with rest of application
Nim—metaprogramming

- **Templates**: in-line code substitutions, also allows overloading, completely hygienic (if desired)

- **Generics**: applies to types, procedures, templates, and macros also allows type-classes, concepts

- **Macros**: similar to Lisp: syntax tree of arguments passed to macro at compile time (type checked or untyped)

- **AST based overloading**: allows specialization based on the AST of the arguments
New framework: QEX (Quantum EXpressions)

- Data parallel library for tensor objects on a lattice including shifts, reductions
- Mostly in Nim, with USQCD SciDAC C libraries
- High level interface in development
- Available on https://github.com/jcosborn/qex
- Performance portability study: cudanim
  - Supports arrays on both CPU and GPU
  - Checkout https://github.com/jcosborn/cudanim
onGpu

1: template onGpu**(nn, tpb: untyped, body: untyped): untyped =  
2:  block:  
3:   var v = packVars(body, getGpuPtr)  
4:   type ByCopy { .bycopy. } [T] = object  
5:     d: T  
6:   proc kern(xx: ByCopy[type(v)]) {.cudaGlobal.} =  
7:     template deref(k: int): untyped = xx.d[k]  
8:     substVars(body, deref)  
9:   let ni = nn.int32  
10:  let threadsPerBlock = tpb.int32  
11:  let blocksPerGrid = (ni+threadsPerBlock-1) div threadsPerBlock  
12:  cudaLaunch(kern, blocksPerGrid, threadsPerBlock, v)  
13:  discard cudaDeviceSynchronize()  
14: template onGpu**(nn: untyped, body: untyped): untyped =  
15:  onGpu(nn, 64, body)  
16: template onGpu**(body: untyped): untyped =  
17:  onGpu(512*64, 64, body)
The generated kern

1: proc kern(xx: ByCopy[type(v)])
2:   {.codegenDecl: "__global__ $# $#$$.} = # Some definitions omitted
13: inlineProcs: Inlines all procedure calls
14:   template deref(k: int): untyped =
15:     xx.d[k]
16:   substVars((x += y * z), deref)

• xx is an object holding pointers to GPU memory
• substVars with the help of deref transforms

x += y * z  →  deref(0)+=deref(1)*deref(2)
Coalesced in-memory data layout

AoSoAoS
Tesla P100 + 2x Xeon E5-2687WV2

\[ X_{ij}^{l} \leftarrow X_{ij}^{l} + Y_{ik}^{l} \times Z_{kj}^{l} \]

CUDA 8.0.61 + GCC 4.8.5
KNL (Xeon Phi 7210)  
Flat MCDRAM  

$X_{ij}^l \leftarrow X_{ij}^l + Y_{ik}^l \times Z_{kj}^l$

GCC 7.1.0
Summary & Outlook

- Nim metaprogramming helps hiding architecture differences under a unified data parallel API
- Toy benchmark saturates GPU bandwidth
- Considering API to use both CPU & GPU in a heterogeneous setting
- Many possibilities of using AST
  - Apply AST based optimizations (inlining, loop unrolling, temporary variable elimination), across multiple statements
  - Specialization at different levels
  - Craft application specific AST transformations
  - Help general purpose compiler with application specifics
Backup slides
GPU offloading (1 of 4)

1: template cudaDefs(body: untyped): untyped {.dirty.} =
2:   var blockDim{.global,importC,noDecl.}: CudaDim3
3:   var blockIdx{.global,importC,noDecl.}: CudaDim3
4:   var blockDim{.global,importC,noDecl.}: CudaDim3
5:   var threadIdx{.global,importC,noDecl.}: CudaDim3
6:   template getGridDim: untyped {.used.} = blockDim
7:   template getBlockIdx: untyped {.used.} = blockIdx
8:   template getBlockSize: untyped {.used.} = blockDim
9:   template getThreadIdx: untyped {.used.} = threadIdx
10:  template getThreadNum: untyped {.used.} = blockDim.x * blockDim.x + threadIdx.x
11:  template getNumThreads: untyped {.used.} = blockDim.x * blockDim.x
12:  bind inlineProcs
13:  inlineProcs:
14:    body

- Overloaded template definitions
- Inline all Nim procedures in **body** to prepare for generating kernel function
· Convert a procedure definition, p, with the overloaded templates in cudaDefs

· Turn it in to a __global__ kernel
GPU offloading (3 of 4)

16: **template** `cudaLaunch*`(p: proc; blocksPerGrid,threadsPerBlock: SomeInteger;
17: arg: varargs[pointer,dataAddr]) =
18: var pp: proc = p
19: var gDim, blockDim: CudaDim3
20: gDim.x = blocksPerGrid
21: gDim.y = 1
22: gDim.z = 1
23: blockDim.x = threadsPerBlock
24: blockDim.y = 1
25: blockDim.z = 1
26: var args: array[arg.len, pointer]
27: for i in 0..<arg.len: args[i] = arg[i]
28: #echo "really launching kernel"
29: let err = cudaLaunchKernel(pp, gDim, blockDim, addr args[0])
30: if err:
31: echo err
32: quit cast[cint](err)

- Calls the cuda function, cudaLaunchKernel, with a passed in procedure, p
GPU offloading (4 of 4)

48: template onGpu*(nn, tpb: untyped, body: untyped): untyped =
49:     block:
50:     var v = packVars(body, getGpuPtr)
51:     type ByCopy {.bycopy.} [T] = object
52:     d: T
53:     proc kern(xx: ByCopy[type(v)]) {.cudaGlobal.} =
54:         template deref(k: int): untyped = xx.d[k]
55:         substVars(body, deref)
56:     let ni = nn.int32
57:     let threadsPerBlock = tpb.int32
58:     let blocksPerGrid = (ni+threadsPerBlock-1) div threadsPerBlock
59:     cudaLaunch(kern, blocksPerGrid, threadsPerBlock, v)
60:     discard cudaDeviceSynchronize()
61: template onGpu*(nn: untyped, body: untyped): untyped = onGpu(nn, 64, body)
62: template onGpu*(body: untyped): untyped = onGpu(512*64, 64, body)

- Take a body of code chunk and put it in a kernel definition, kern

- kern calls cudaGlobal to setup other definitions, and takes care of syncing CPU memory to GPU