# 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

$$X^l_{ij} \leftarrow X^l_{ij} + Y^l_{ik} \times Z^l_{kj}$$

---

```
proc test(vecLen, memLen: static[int]; N: int) =
  var
    x = newColorMatrixArray(vecLen,memLen,N)
    y = newColorMatrixArray(vecLen,memLen,N)
    z = newColorMatrixArray(vecLen,memLen,N)
```

Define complex 3×3 matrix field

```
  threads:
```

CPU threads

```
    x := 0
    y := 1
    z := 2
```

Set diagonal elements

```
    timeit "CPU": x += y * z
```

Benchmark

```

  timeit "GPU5":
    onGpu(N, 32):
      x += y * z
```

Run statement block on GPU

```
  timeit "GPU6": onGpu(N, 64): x += y * z
  timeit "GPU7": onGpu(N, 128): x += y * z
```

Rerun with different T/B

```

  threads: timeit "CPU": x += y * z
```

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 * z` → `x[i] += (y * z)[i]`

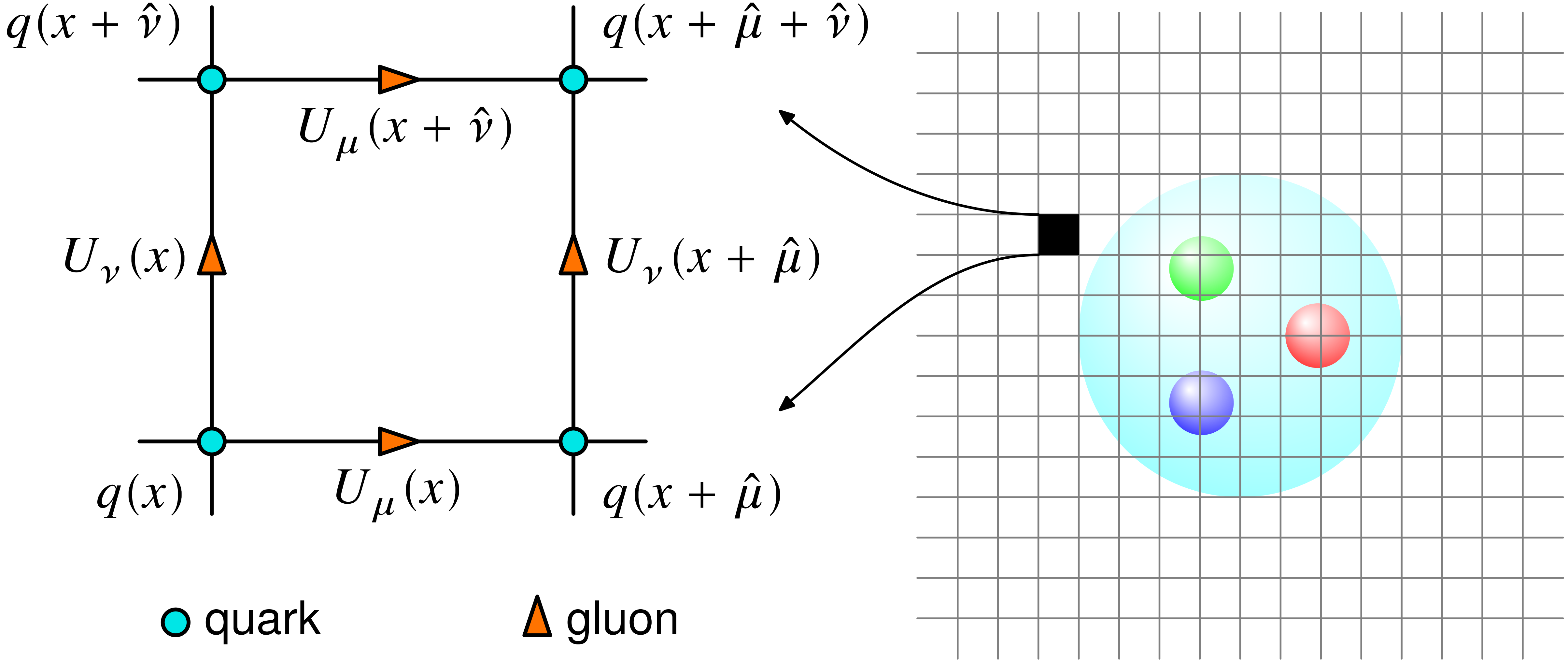
# AST based overloading for data parallel ops

```
type ArrayIndex* = SomeInteger or ShortVectorIndex

template indexArray*(x: ArrayObj, i: ArrayIndex): untyped =
  x.p[i]

macro indexArray*(x: ArrayObj{call}, y: ArrayIndex): untyped =
  result = newCall(ident($x[0]))
  for i in 1..<x.len:
    let xi = x[i]
    result.add( quote do:
      indexArray(`xi`,`y`) )

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

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

`x[i] += (y * z)[i]` → `x[i] += y[i] * z[i]`

# Lattice gauge theory



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

# 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

# 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

```
template onGpu*(nn,tpb: untyped, body: untyped): untyped =
  block:
    var v = packVars(body, getGpuPtr)
    type ByCopy {.bycopy.} [T] = object
      d: T
    proc kern(xx: ByCopy[type(v)]) {.cudaGlobal.} =
      template deref(k: int): untyped = xx.d[k]
      substVars(body, deref)
    let ni = nn.int32
    let threadsPerBlock = tpb.int32
    let blocksPerGrid = (ni+threadsPerBlock-1) div threadsPerBlock
    cudaLaunch(kern, blocksPerGrid, threadsPerBlock, v)
    discard cudaDeviceSynchronize()
template onGpu*(nn: untyped, body: untyped): untyped =
  onGpu(nn, 64, body)
template onGpu*(body: untyped): untyped =
  onGpu(512*64, 64, body)
```

Copy memory to GPU
if necessary
collect pointers in v

Generate kernel function

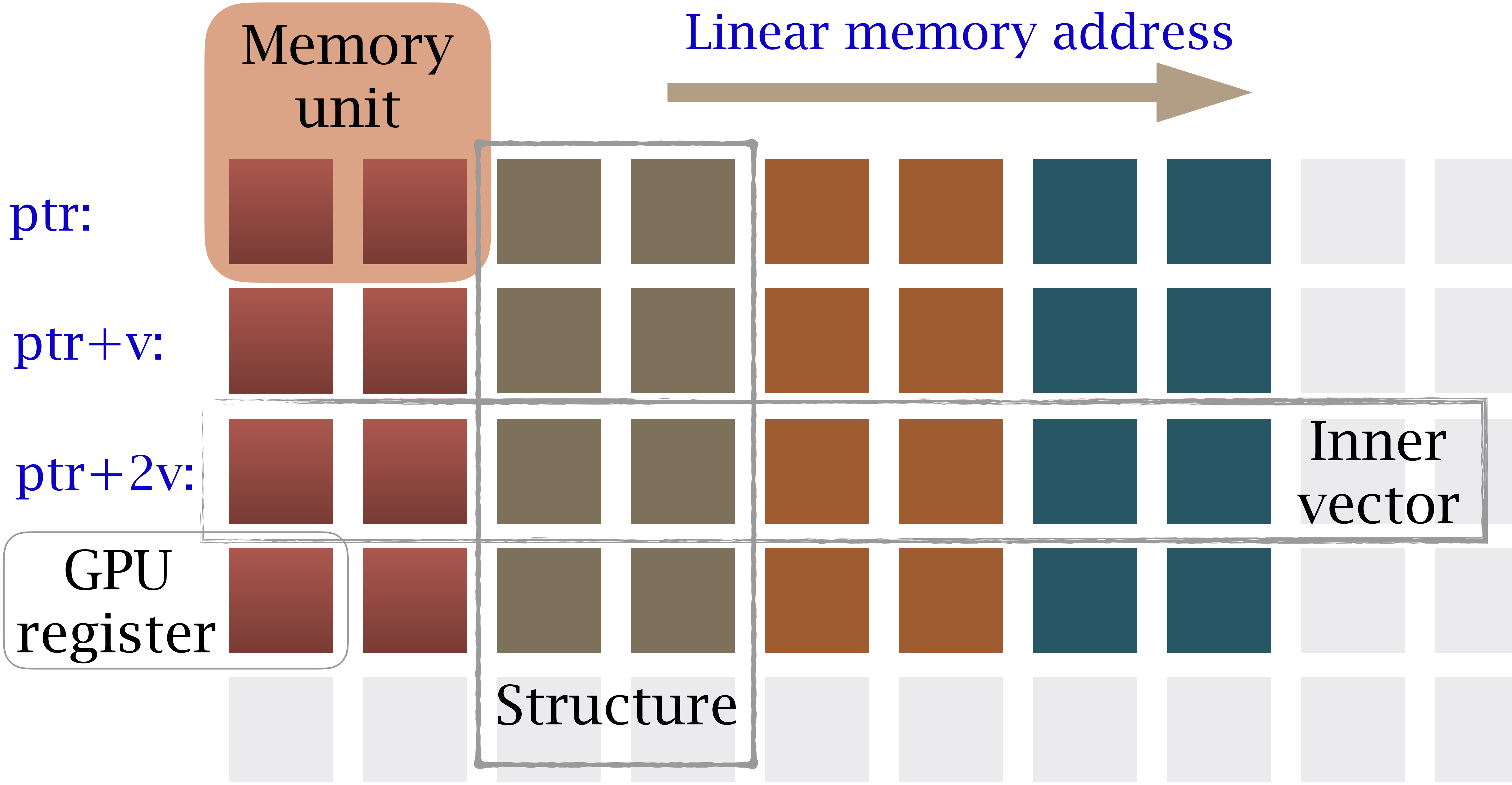
# The generated `kern`

```
proc kern(xx: ByCopy[type(v)])
     {.codegenDecl: "__global__ $# $#$#".} =
  # Some definitions omitted
  inlineProcs:
    template deref(k: int): untyped =
      xx.d[k]
    substVars((x += y * z), deref)
```

Inlines all procedure calls

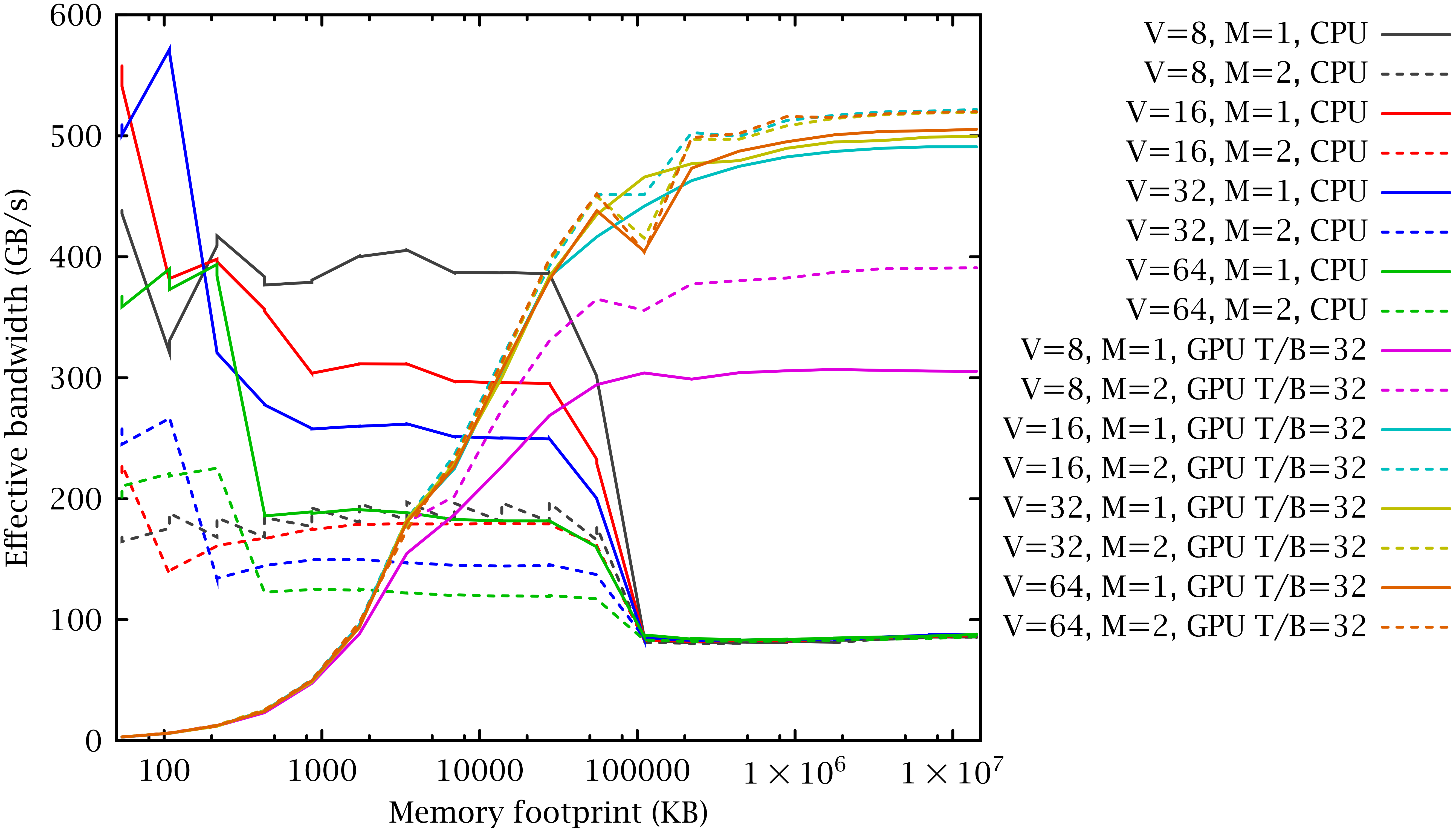
- `xx` is an object holding pointers to GPU memory
- `substVars` with the help of `deref` transforms

```
x += y * z → deref(0)+=deref(1)*deref(2)

           → xx.d[0]+=xx.d[1]*xx.d[2]
```



# Coalesced in-memory data layout

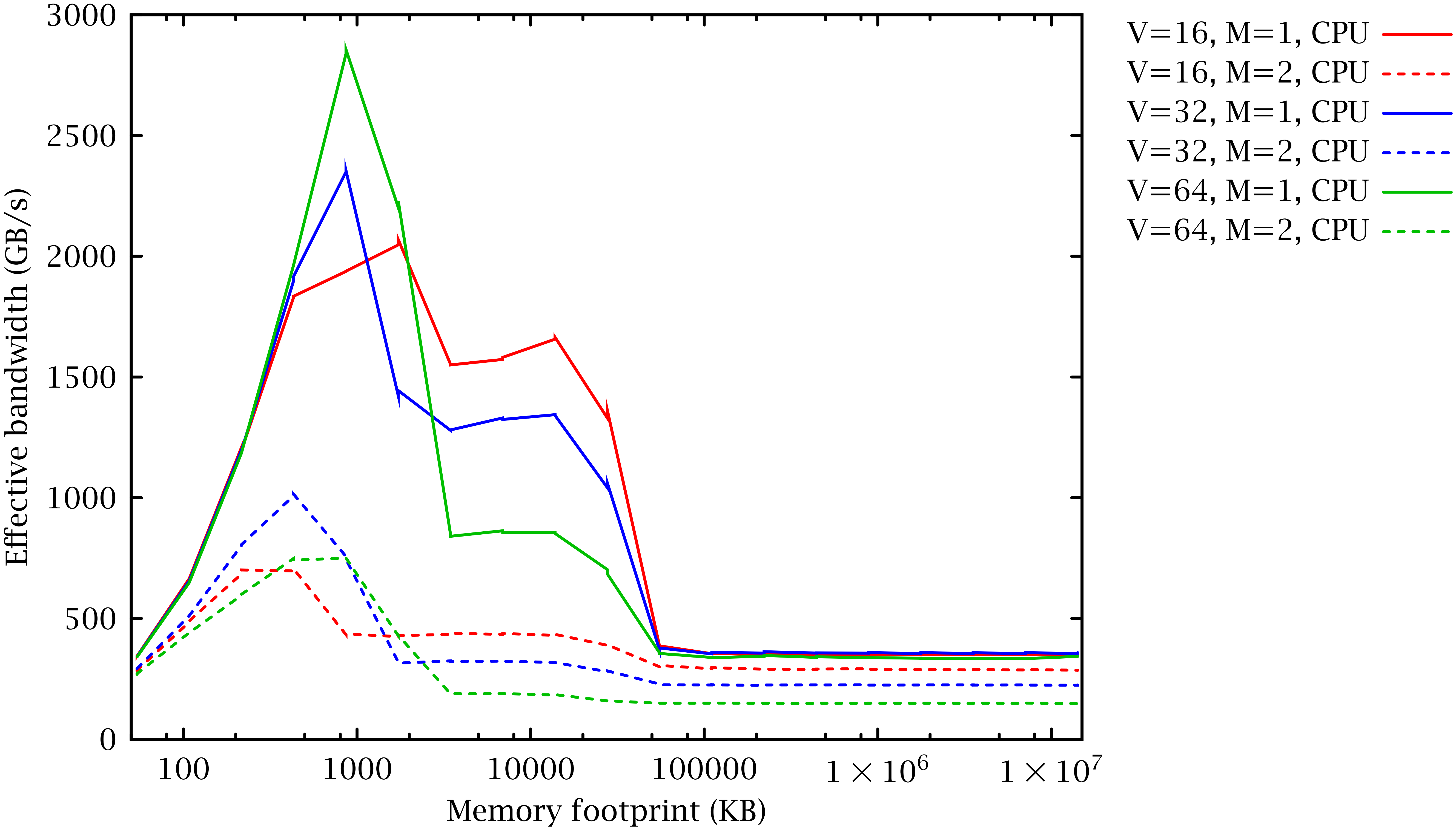
AoSoAoS



# Tesla P100 +
# 2x Xeon E5-2687WV2

$$X^l_{ij} \leftarrow X^l_{ij} + Y^l_{ik} \times Z^l_{kj}$$

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)

```
template cudaDefs(body: untyped): untyped {.dirty.} =
  var gridDim{.global,importC,noDecl.}: CudaDim3
  var blockIdx{.global,importC,noDecl.}: CudaDim3
  var blockDim{.global,importC,noDecl.}: CudaDim3
  var threadIdx{.global,importC,noDecl.}: CudaDim3
  template getGridDim: untyped {.used.} = gridDim
  template getBlockIdx: untyped {.used.} = blockIdx
  template getBlockDim: untyped {.used.} = blockDim
  template getThreadIdx: untyped {.used.} = threadIdx
  template getThreadNum: untyped {.used.} = blockDim.x * blockIdx.x + threadIdx.x
  template getNumThreads: untyped {.used.} = gridDim.x * blockDim.x
  bind inlineProcs
  inlineProcs:
    body
```

- Overloaded template definitions
- Inline all Nim procedures in `body` to prepare for generating kernel function

# GPU offloading (2 of 4)

```
macro cuda*(s,p: untyped): auto =
  let ss = s.strVal
  p.expectKind nnkProcDef
  result = p
  result.addPragma parseExpr("{.codegenDecl:\""&ss&" $# $#$#\".}")[0]
  result.body = getAst(cudaDefs(result.body))
  var sl = newStmtList()
  sl.add( quote do:
    {.push checks: off.}
    {.push stacktrace: off.} )
  sl.add result
  result = sl
template cudaGlobal*(p: untyped): auto = cuda("__global__",p)
```

- Convert a procedure definition, `p`, with the overloaded templates in `cudaDefs`

- Turn it in to a `__global__` kernel

# GPU offloading (3 of 4)

```
template cudaLaunch*(p: proc; blocksPerGrid,threadsPerBlock: SomeInteger;
                     arg: varargs[pointer,dataAddr]) =
  var pp: proc = p
  var gridDim, blockDim: CudaDim3
  gridDim.x = blocksPerGrid
  gridDim.y = 1
  gridDim.z = 1
  blockDim.x = threadsPerBlock
  blockDim.y = 1
  blockDim.z = 1
  var args: array[arg.len, pointer]
  for i in 0..<arg.len: args[i] = arg[i]
  #echo "really launching kernel"
  let err = cudaLaunchKernel(pp, gridDim, blockDim, addr args[0])
  if err:
    echo err
    quit cast[cint](err)
```

- Calls the cuda function, cudaLaunchKernel, with a passed in procedure, p

# GPU offloading (4 of 4)

```
template onGpu*(nn,tpb: untyped, body: untyped): untyped =
  block:
    var v = packVars(body, getGpuPtr)
    type ByCopy {.bycopy.} [T] = object
      d: T
    proc kern(xx: ByCopy[type(v)]) {.cudaGlobal.} =
      template deref(k: int): untyped = xx.d[k]
      substVars(body, deref)
    let ni = nn.int32
    let threadsPerBlock = tpb.int32
    let blocksPerGrid = (ni+threadsPerBlock-1) div threadsPerBlock
    cudaLaunch(kern, blocksPerGrid, threadsPerBlock, v)
    discard cudaDeviceSynchronize()
template onGpu*(nn: untyped, body: untyped): untyped = onGpu(nn, 64, body)
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