A Software Framework in Python for Generating Optimal Isogeometric Kernels on the PowerPC 450

Aron Ahmadia[1], *Jed Brown*[2], Nathan Collier[1], Tareq Malas[1], John Gunnels[3]

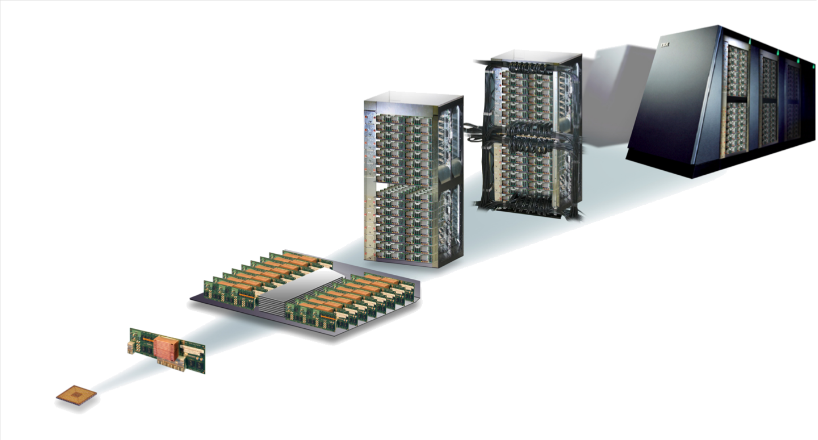
[1]King Abdullah University of Science and Technology

[2]Argonne National Laboratory / ETH Zürich

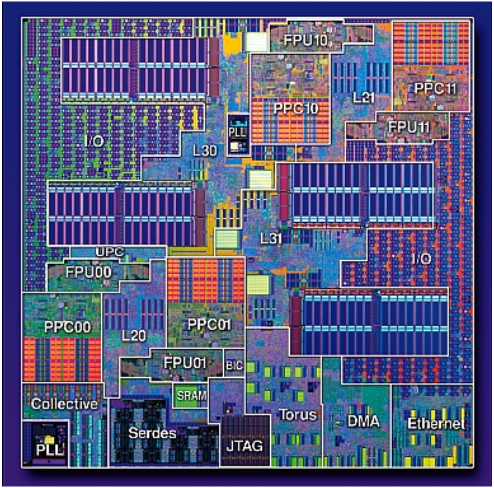
[3]IBM Watson

2011-07-13

Blue Gene/P



Blue Gene/P



- 4 cores @ 850 Mhz
- 32 16-bytes FP registers
- 1 packed FMA per cycle, latency 5
- 0.5 load per cycle, latency 4
- 3 memory requests in-flight
- write-through cache, FIFO eviction policy
- up to 5 memory streams

What makes compiled code slow?

Compilers are bad at

- SIMD instructions
- Alignment constraints
- Register allocation
- Scheduling for out-of-order execution
- Transformations to reduce memory bandwidth

But it's not hopeless

- BG/P has rich SIMD instructions
- Large kernels reuse small kernels
- Register allocation usually has a pattern

Code generation

Mako templates writing inline assembly

- Easy to control unrolling and jamming
- Hard to manage generators with complex control flow
- Hard to keep track of register names and debug
- How to manage in-order execution?
- Smells bad

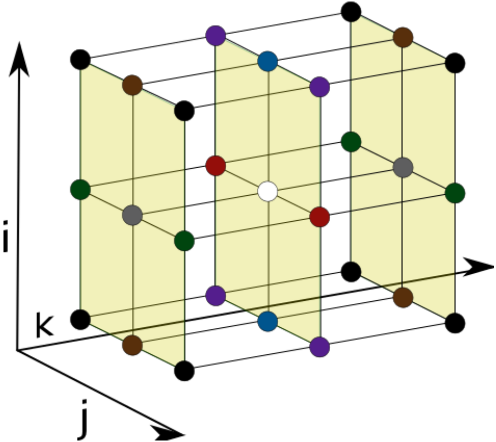
SimASM: All Python

- Name some or all registers, can mix pinned and unpinned registers
- Build kernel using generators/loops/objects/etc
- Transform to partial order according to instruction dependencies (hazards)
- Transform/traverse using simulator, can debug correctness too
- Hazards that cause stalls are shown and explained

Instruction Set

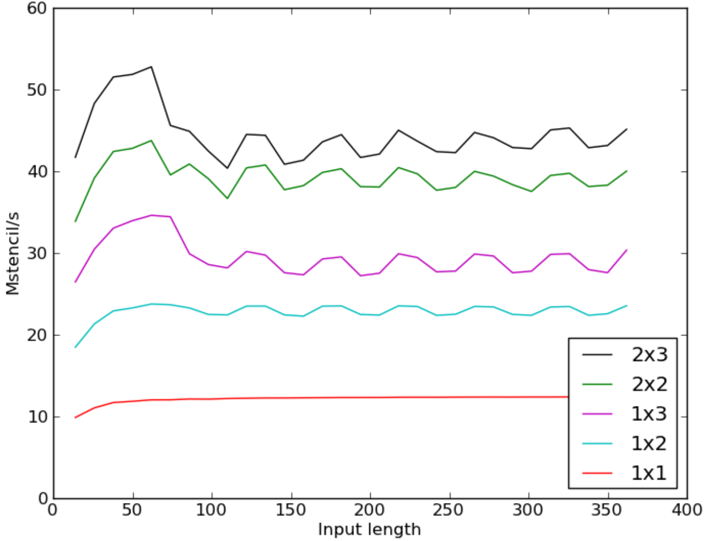
```
class fxcxma(Instruction):
  def __init__(self,rt,ra,rc,rb):
    Instruction.__init__(self)
    self.save(locals(),'rt ra rc rb')
    self.reads(ra,rc,rb)
    self.writes(rt)
    self.uses(PPC.FP,5)
  def run(self,c):
    ra,rc,rb = c.access_fpregisters(self.ra,self.rc,self.rb)
    c.fp[c.get_fpregister(self.rt)] =
                                           FPVal(ra.s*rc.s + rb.p,
                                                 ra.s*rc.p + rb.s)
```

Stencil Operation



- Cartesian grid, constant coefficient scalar PDE.
- Forward propagation operator or Jacobi smoother.
- Memory bandwidth limited? (Datta et al. 2009, SIAM Review)
  - Cache blocking: 26 Mstencil/s (41% of theoretical 63 at FPU peak)
- Load/store and FPU limited?
  - Jamming and SIMD: 93% in L1, 70% from DRAM

Stencil Performance



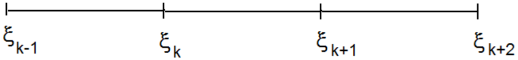
(L2 prefetch cache associativity effects when streaming from DRAM)

Stencil implementation

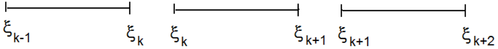
```
# do the FMA's for frame 1/3
for i in self.block_ind:
  istream += self.fma_block(com.w, com.streams,
                            com.results, i, self.K0)

# mute for frame 2/3
istream += [
  isa.lfsdux(com.streams[i],com.a_ptr,com.a_indexing[i])
                             for i in range(self.FRAME_SIZE)]

# do the FMA's for frame 2/3
for i in self.block_ind:
  istream += self.fma_block(com.w, com.streams,
                 com.results, i, self.K1)
```

- Variable blocking and jamming, no need to worry about scheduling.
- Can build special-purpose vectorized primitives (`fma_block`)
- No need to worry about instruction dependencies.

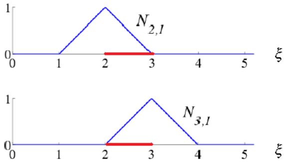
Isogeometric finite elements

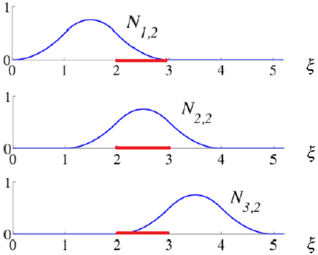


Partition mesh into elements (non-zero knot spans)



There are p+1 functions of order *p* assigned to an element $K = [\ \xi_k, \xi_{k+1}\ ]$

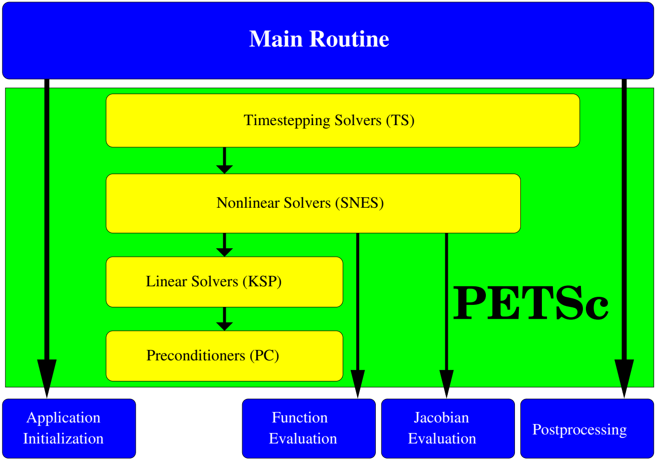




Given knot numbers and order suffices to compute all relevant degree-of-freedom interactions in 1D, 2D and 3D

IGA compared to standard FEM

- Can exactly conform to some engineering geometries.
- Better impedence match with solid modeling (CAD).
- Fewer degrees of freedom for 4th order problems,
  e.g. no rotation dofs for shells.
- More nonzeros per row as continuity is increased.
- More quadrature points per dof (higher arithmetic intensity).
- Needs logically structured grids
  (T-splines can join structured patches)
- All-positive basis functions useful for some problems
  (maintain positivity, robust conservative normals)
- Non-interpolatary basis can be tricky for preconditioning.

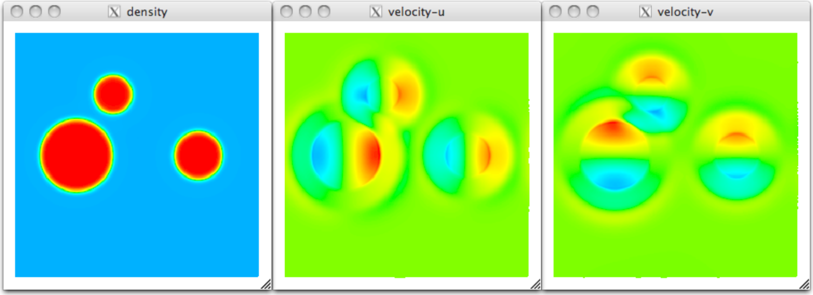


- IGA used to evaluate nonlinear residuals
- PETSc `DA` used to manage parallelism.
- Adaptive time integration using method of lines.
  - Generalized α method from PETSc `TS`.
- Matrix-free Newton-Krylov, need only residuals for implicit solve.

Navier-Stokes Korteweg

Phase field model for water/water vapor two-phase flows. Find $U = (\rho, u)$ such that $B(W,U) = 0$ for all $W = (q, w)$, plus boundary conditions.

$$B(W,U) = \int_\Omega q\frac{\partial \rho}{\partial t} - \nabla q \cdot \rho u + w \cdot \left[u\frac{\partial \rho}{\partial t} + \rho\frac{\partial u}{\partial t} \right]$$
$$+ \nabla w : \left[-\rho u \otimes u + \tau - (p + \lambda |\nabla \rho|^2) 1 \right]$$
$$- \nabla(\nabla \cdot w) \cdot \lambda \rho \nabla \rho - \nabla(\nabla \rho \cdot w) \cdot \lambda \nabla \rho = 0$$



Navier-Stokes Korteweg

Phase field model for water/water vapor two-phase flows. Find $U = (\rho, u)$ such that $B(W,U) = 0$ for all $W = (q, w)$, plus boundary conditions.

$$
\begin{aligned}
B(W,U) = \int_\Omega q \frac{\partial \rho}{\partial t} - \nabla q \cdot \rho u + w \cdot \left[u \frac{\partial \rho}{\partial t} + \rho \frac{\partial u}{\partial t} \right] \\
+ \nabla w : \left[-\rho u \otimes u + \tau - (p + \lambda \left|\nabla \rho\right|^2)) 1 \right] \\
- \nabla(\nabla \cdot w) \cdot \lambda \rho \nabla \rho - \nabla(\nabla \rho \cdot w) \cdot \lambda \nabla \rho = 0
\end{aligned}
$$

```
for each Na,Na_x,Na_xx,Na_y,Na_yy: // test functions
  R_rho   = Na*rho_t;
  R_rho += -rho*(Na_x*ux + Na_y*uy);
  R_ux   = Na*ux*rho_t;
  R_ux += Na*rho*ux_t;
  R_ux += -rho*(Na_x*ux*ux + Na_y*ux*uy);
  R_ux += -Na_x*p;
  R_ux += rRe*(Na_x*tau_xx + Na_y*tau_xy);
  R_ux += -Ca2*rho*(Na_xx*rho_x + Na_xy*rho_y);
  ...
```

Transform to more vector-friendly form

- Pre-compute "physics" W at each quadrature point
- assembling the residual becomes dot products

```
for each Na,Na_x,Na_xx,Na_y,Na_yy:
  R_rho  = Na*W[irho_t];
  R_rho += Na_x*W[rho_nax];
  R_rho += Na_y*W[rho_nay];
  R_ux  = Na*W[ux_na];
  R_ux += Na_x*W[ux_nax];
  R_ux += Na_y*W[ux_nay];
  R_ux += Na_xx*W[u_naxx];
  R_ux += Na_xy*W[u_naxy];
```

- 1.9x speedup

Vectorize using SimASM

- Define context-sensitive vector primitives

```
def muladd_copy(self, com, rt, ra, rb):
  if ra[1] == 0:
    return isa.fxcpmadd(rt,com.W[ra[0]],rb,rt)
  else:
    return isa.fxcsmadd(rt,com.W[ra[0]],rb,rt)
```

- Unrolled/jammed vector assembly looks “close” to the physics

```
[self.muladd_copy(com, 'R_rho', com.rho_nax, 'Na_x'),
self.muladd_copy(com, 'R_ux', com.ux_nax, 'Na_x'),
self.muladd_copy(com, 'R_uy', com.uy_nax, 'Na_x')]
```

- Still limited by load/store unit.
- Multiple quadrature points and elements could amortize load/store cost.
- More clever transformations?
- Still need to optimize computation of coordinate transformation for high end-to-end throughput.

Perspective on SimASM

Blue Gene/P is representative of future architectures

- In-order execution
- Longer FP registers
- More cores
- Less memory bandwidth

Need some way to get close to peak performance

- SSE intrinsics are pretty good on Intel/AMD
  - Better designed intrinsic API
  - Out of order execution more tolerant
  - Fewer registers
  - Lightweight templating (e.g. Mako) might be good enough
- Interesting alternatives
  - OpenCL (wide vectorization, different memory model)
  - Intel SPMD Program Compiler (`ispc.github.com`)

Outlook

Lots more to do with IGA/FEM

- Library interface for vectorized physics/assembly
- Connecting structured blocks (T-splines)
- Algorithmic (analytic Jacobians, preconditioning)

SimASM

- Better optimization framework.
- Different target architectures (e.g. Blue Gene/Q, Knight's Corner).
- Interface improvements/visualization.
- Code generation from high level/symbolic description?
- `bitbucket.org/jedbrown/simasm`