MetaMorph

A Modular Library of Malleable Accelerator Primitives for Heterogeneous Parallel Computing

SEEC Fall Symposium 2015

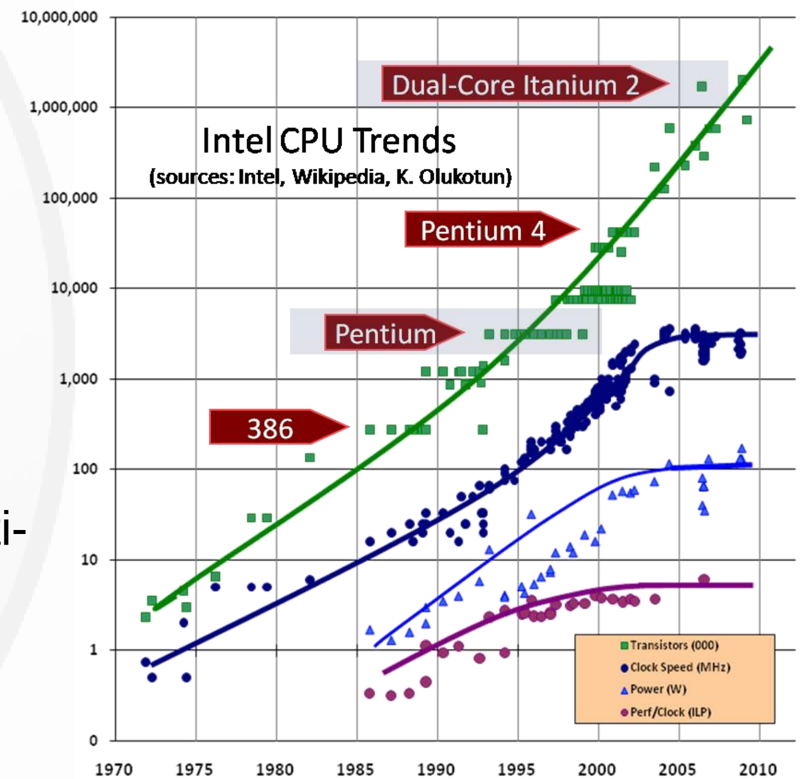
Paul Sathre, **Ahmed Helal**

VirginiaTech *Invent the Future*

SyNeRG synergy.cs.vt.edu

Motivation:

- Why do we need accelerator?
  - Free lunch is over!
  - You can't anymore wait until H/W vendors improve your performance with faster CPUs.



- However, parallel programming is complex!
  - Architectures: Multicore CPUs, multi-socket cc-NUMA, multi-nodes, GPUs, MIC,…
  - Programming models: OpenMP, OpenACC, MPI, CUDA, OpenCL,…
  - *Five years from now, you will NOT know what hardware will look like*

Motivation:

Productivity = Programmability + Performance + Portability

- Dot Product (Serial on CPU)

HOST

Serial C

```
void dotProd(double *a,
             double *b,
             double *ret,
             int n_elem)
{
  int i;
  *ret = 0.0;
  for (i = 0; i < n_elem; i++)
    ret += a[i] * b[i];
}
```

10

DEVICE

Motivation:

Productivity = Programmability + Performance + Portability

- Dot Product (Parallel on GPU using CUDA)

HOST

27

```
int main(int argc, char ** argv)
{
  int n_elem;
  double *a, *b, *ret, sum = 0.0;
  //allocate and initialize a, b, and ret
  ...
  double *dev_a, *dev_b, *dev_r;
  size_t size = n_elem * sizeof(double);
  //allocate device buffers
  cudaMalloc((void**)&dev_a, size);
  cudaMalloc((void**)&dev_b, size);
  cudaMalloc((void**)&dev_r, size);

  //initialize device buffers
  cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
  cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

  //set grid/block size
  ...
  //multiply elements
  dotProd-vmul<<<grid, block>>>(dev_a, dev_b, dev_r,
    n_elem);

  //partial result
  cudaMemcpy(ret, dev_r, size, cudaMemcpyDeviceToHost);

  //CPU sum
  int i;
  for (i = 0; i < n_elem; i++)
    sum += ret[i];
}
```

DEVICE (GPU)

```
__global void dotProd-vmul(
    double *a, double *b,
    double *ret, int n_elem)
{
  int i;
  int tid = blockIdx.x *
    blockDim.x + threadIdx.x;
  int n_threads = blockDim.x *
    gridDim.x;
  for (i=tid; i<n_elem; i+=n_threads)
    ret[tid] = a[tid] * b[tid];
}
```

Motivation:

Productivity = Programmability + Performance + Portability

- Dot Product (Optimized Parallel on GPU Device)

HOST **39**

```
int main(int argc, char **argv)
{
  //global and block sizes
  int ni, nj, nk, tx, ty, tz;
  //pick a node and zeroth device
  cudaSetDevice(0)
  //declare host memory
  double *a, *b, *ret, zero = 0.0;
  //allocate and initialize it
  …

  //Allocate device buffers
  size_t size = sizeof(double)*ni*nj*nk;
  double *dev_a, *dev_b, *dev_r;
  cudaMalloc(&dev_a, size);
  cudaMalloc(&dev_b, size);
  cudaMalloc(&dev_r, sizeof(double));
  //initialize them
  cudaMemcpy(dev_a, a, size,
    cudaMemcpyHostToDevice);
  cudaMemcpy(dev_b, a, size,
    cudaMemcpyHostToDevice);
  cudaMemcpy(dev_r, &zero, sizeof(double),
    cudaMemcpyHostToDevice);

  //set computation shape
  dim3 grid, block, shape, start, end;
  block = dim3(tx, ty, tz);
  grid = dim3((ni+tx-1)/ni, (nj+ty-1)/ty, 1);
  shape = dim3(ni, nj, nk);
  start = dim3(0, 0, 0);
  end = dim3(ni-1, nj-1, nk-1);

  //run the kernel
  dotProd<<<grid, block, tx*ty*tz*sizeof(double)>>>
    (dev_a, dev_b, shape.x, shape.y, shape.z,
    start.x, start.y, start.z, end.x, end.y, end.z,
    (nk+tz-1)/tz, dev_r, tx*ty*tz);

  //bring the dot product back
  cudaMemcpy (ret, dev_r, sizeof(double),
    cudaMemcpyDeviceToHost);
}
```

DEVICE (GPU) **40**

```
__device__ void block_reduction(double *psum,
  int tid, int len_)
{
  int stride = len_ >> 1;
  while (stride > 0) {
    if (tid  < stride)
      psum[tid] += psum[tid+stride];
    __syncthreads();
    stride >>= 1;
  }
}

//Implementation of double atomicAdd
...

__global__ void kernel_dotProd(double *phi1, double *phi2,
int i, int j, int k, int sx,
  int sy, int sz, int ex, int ey, int ez,
  int gz, T * reduction, int len_)
{
  extern shared psum[];
  int tid, x, y, z, itr;
  bool boundx, boundy, boundz;
  tid = threadIdx.x + (threadIdx.y) * blockDim.x
    + (threadIdx.z) * (blockDim.x * blockDim.y);
  x = (blockIdx.x)*blockDim.x+threadIdx.x+sx;
  y = (blockIdx.y)*blockDim.y+threadIdx.y+sy;

  psum[tid] = 0;
  boundy = ((y >= sy) && (y <= ey));
  boundx = ((x >= sx) && (x <= ex));

  for (itr = 0; itr < gz; itr++) {
    z = itr*blockDim.z+threadIdx.z +sz;
    boundz = ((z >= sz) && (z <= ez));
    if (boundx && boundy && boundz)
      psum[tid] += phi1[x+y*i+z*i*j] *
        phi2[x+y*i+z*i*j];
  }
  __syncthreads();
  block_reduction(psum,tid,len_);
  __syncthreads();

  if(tid == 0)
    atomicAdd(reduction,psum[0]);
}
```

Motivation:

- What is MetaMorph?
  - A runtime library that hides the complexity of accelerator programming and interoperability behind a standard interface (static API)
    - The static API acts as an interoperability layer between user applications and community-tuned accelerator backends
    - Performance portability still reliant on humans tuning interchangeable backends to specific devices!
- Why do we need it?
  - Extracting the full performance of heterogeneous systems is non-trivial and requires architecture expertise
  - Architectures change faster than codes and scientists shouldn't have to waste their time (re)learning and rewriting code for them

Motivation:

Productivity = Programmability + Performance + Portability

- Dot Product (via our MetaMoph Library)

36

```
int main(int argc, char **argv)
{
  //global and block sizes
  int ni, nj, nk, tx, ty, tz;
  //pick a mode and zeroth device
  choose_accel(0, metaModePreferCUDA);
  //declare host memory
  double *a, *b, *ret, zero = 0.0;
  //allocate and initialize it
  ...

  //Allocate device buffers
  size_t size = sizeof(double)*ni*nj*nk;
  double *dev_a, *dev_b, *dev_r;
  meta_alloc(&dev_a, size);
  meta_alloc(&dev_b, size);
  meta_alloc(&dev_r, sizeof(double));

  //initialize them
  meta_copy_h2d(dev_a, a, size, true);
  meta_copy_h2d(dev_b, a, size, true);
  meta_copy_h2d(dev_r, &zero, sizeof(double),
    true);
  //set computation shape
  a_dim3 grid, block, shape, start, end;
  block[0] = tx, block[1] = ty, block[2] = tz;
  grid[0] = (ni+tx-1)/ni, grid[1] = (nj+ty-1)/ty,
    grid[2] = (nk+tz-1)/tz;
  shape[0] = ni, shape[1] = nj, shape[2] = nk;
  start[0] = 0, start[1] = 0, start[2] = 0;
  end[0] = ni-1, end[1] = nj-1, end[2] = nk-1;

  //run the kernel
  meta_dotProd(&grid, &block, dev_a, dev_b,
    &shape, &start, &end, dev_r, a_db, true);

  //bring the dot product back
  meta_copy_d2h(ret, dev_r, sizeof(double),
    false);

}
```

also support: metaModePreferOpenCL /OpenMP for AMD/MIC/CPU

all kernels/copies can be asynchronous with flag = true, else blocking

grid & block specify thread organization a la CUDA/OpenCL

shape, start, and end allow dot product on arbitrary subregions of 3D space

Goal :

- Support upgrade of legacy CFD codes and development of new codes in heterogeneous environments
  - Many methods:
    - Finite Element, Finite Difference, Finite Volume, etc.
  - Vast range of solvers, preconditioners and other math kernels
  - Range of storage/computation schemes:
    - structured vs. unstructured grids,
    - Sparse vs. Dense storage

- Expand with kernels from other domains
    - Computational Bio, Bigdata, Cosmology, etc.

Desired Features:

- Usable by non-architecture experts (via drop-in replacement functions) and *without* heavyweight framework-centric development

- Current approaches trade portability with heavyweight frameworks
  - provide portability via dynamic remapping of the framework’s data structures to supported devices
  - require redesigning the entire application to use the framework’s complex data structures, which may not easily support operations needed by the application and adds a considerable programming/runtime overhead.

Related Work:

- **OpenFOAM**: a very popular CPU-only CFD simulation framework
- **Paralution**: a new CPU/GPU/MIC BLAS framework
- **MAGMA**: a suite of CUDA, AMD OpenCL, and MIC BLAS libraries
- **Trilinos**: a massive set of multi-physics libraries and frameworks

	MetaMorph	OpenFOAM	Paralution	MAGMA	Trilinos
CFD support	✓	✓	✓	✗	✓
Acceleration support	OMP, CUDA, OCL	No internal support	OMP, CUDA, OCL	OMP, CUDA, OCL	preliminary CUDA/MIC
Target devices	CPU, GPU, MIC	CPU	CPU, GPU, MIC	CPU, GPU, MIC	CPU, GPU, MIC
MPI & interoperability	✓	✗	✓	✗	✗
framework-centric development	✗	✓	✓	✓	✓
External dependencies	✗	✓	✓	✓	✓

Contributions:

- Transparent MPI exchange of accelerator device buffers, including between CUDA and OpenCL
- MetaMorph performance on-par with best language-specific BLAS libraries
- Intelligent Data Marshalling (packing/unpacking) of arbitrary subregions (2D ghost regions) of a 3D structured grid.
- OpenCL context management stack, Fortran and C APIs, event-based timing infrastructure, …

- *I worked on OpenMP backedend for CPU/MIC and OpenCL backedend for MIC*

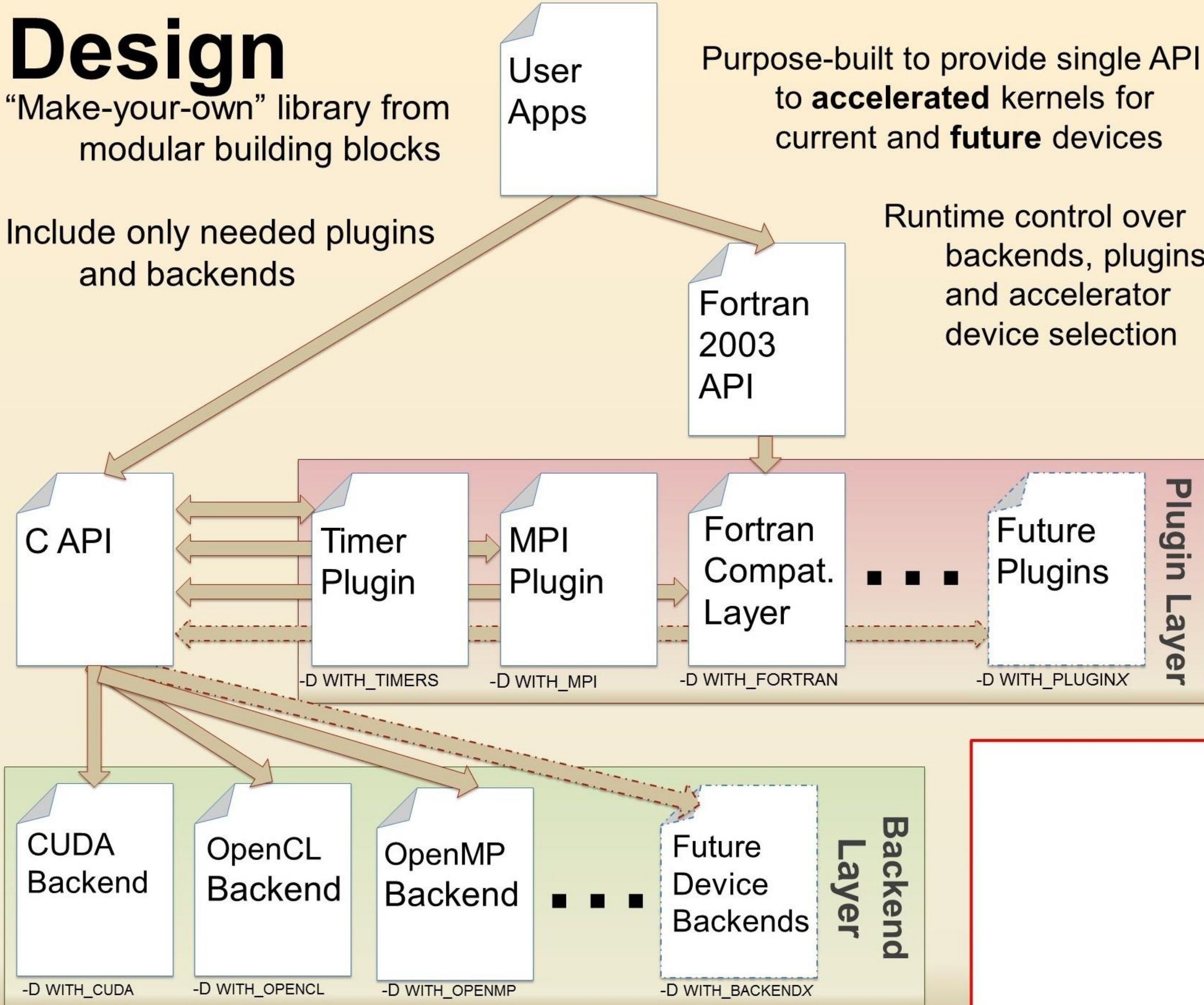
Design

"Make-your-own" library from modular building blocks

Include only needed plugins and backends

Purpose-built to provide single API to **accelerated** kernels for current and **future** devices

Runtime control over backends, plugins, and accelerator device selection



Implementation:

- "Library of Libraries"
  - libmetamorph.so
    - Implements the interface between the top-level MetaMorph API, and the platform-specific backend(s).
  - libmetamorph_x_core.so (x is either OpenMP, CUDA or OpenCL, ..)
    - Set of backend shared library objects ("cores") that provide the actual implementations of the platform-specific kernels.
    - These backends can be custom-tuned for a specific device.

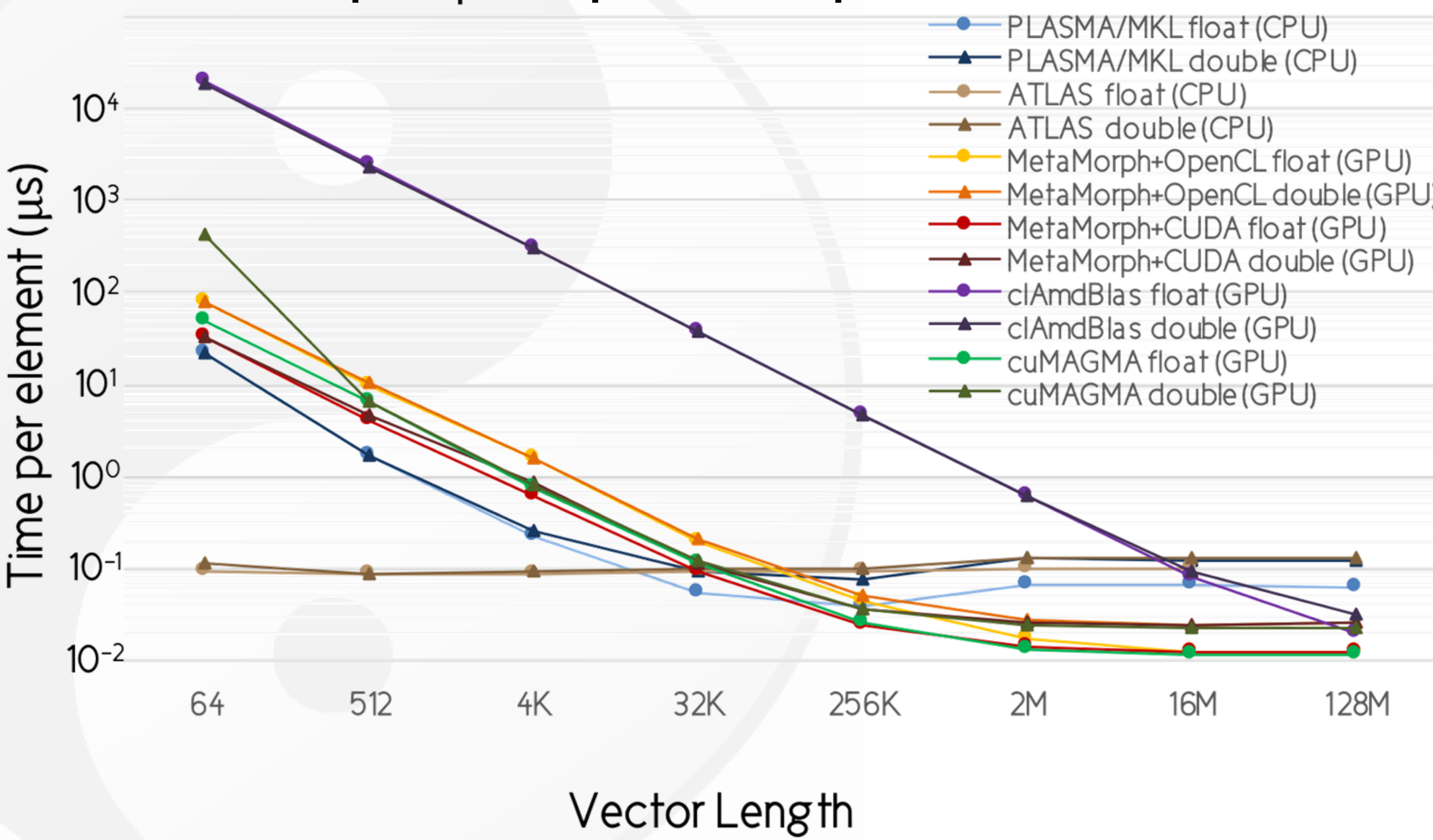
- Compile time:
  - choose which backends to include, choose which (if any) plugins you need from {MPI, Timing, Fortran}

Implementation:

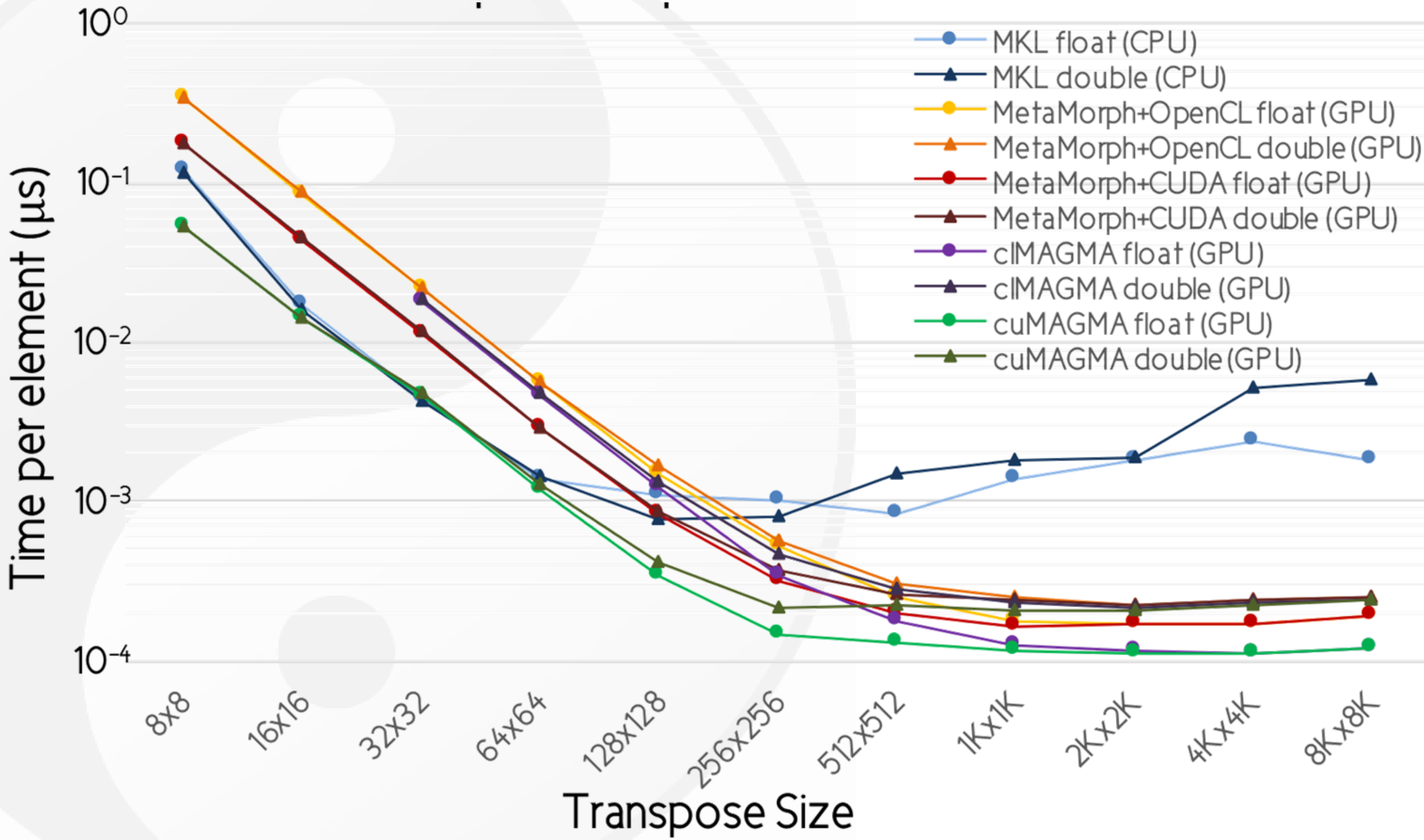
- Runtime:
  - select with environment variables which backend/device are used for a run, timer verbosity, device selection, etc.

- OpenMP backend
  - Default backend
  - To minimize the design modifications, OpenMP backend mimic the accelerator model.
    - There are several unnecessary operations/data movements on CPU.
  - UNIFIED_MEMORY feature
    - User apps can eliminate redundant data transfers.
    - Copy pointers to the data, instead of explicit memory transfer.

MetaMorph: Dot-product performance



MetaMorph: Transpose performance



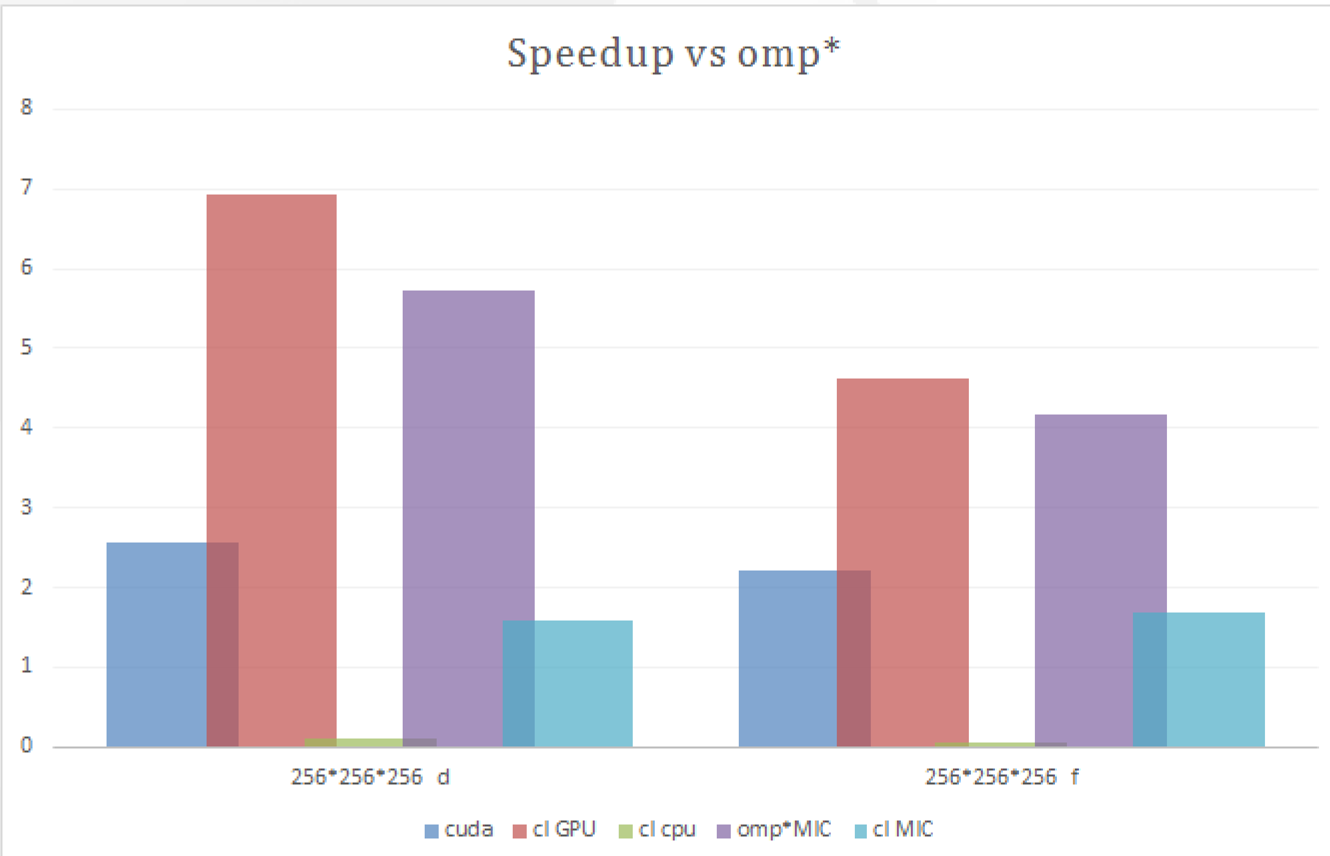
MetaMorph: Library overhead vs. standalone kernels

- OpenMP backend (3D-Dot product with double)
  - Max overhead 5%.

CPU: Intel(R) Core(TM) i5-2400, 4 cores @ 3.10GHz

MetaMorph: Performance vs. CPU OpenMP+AVX

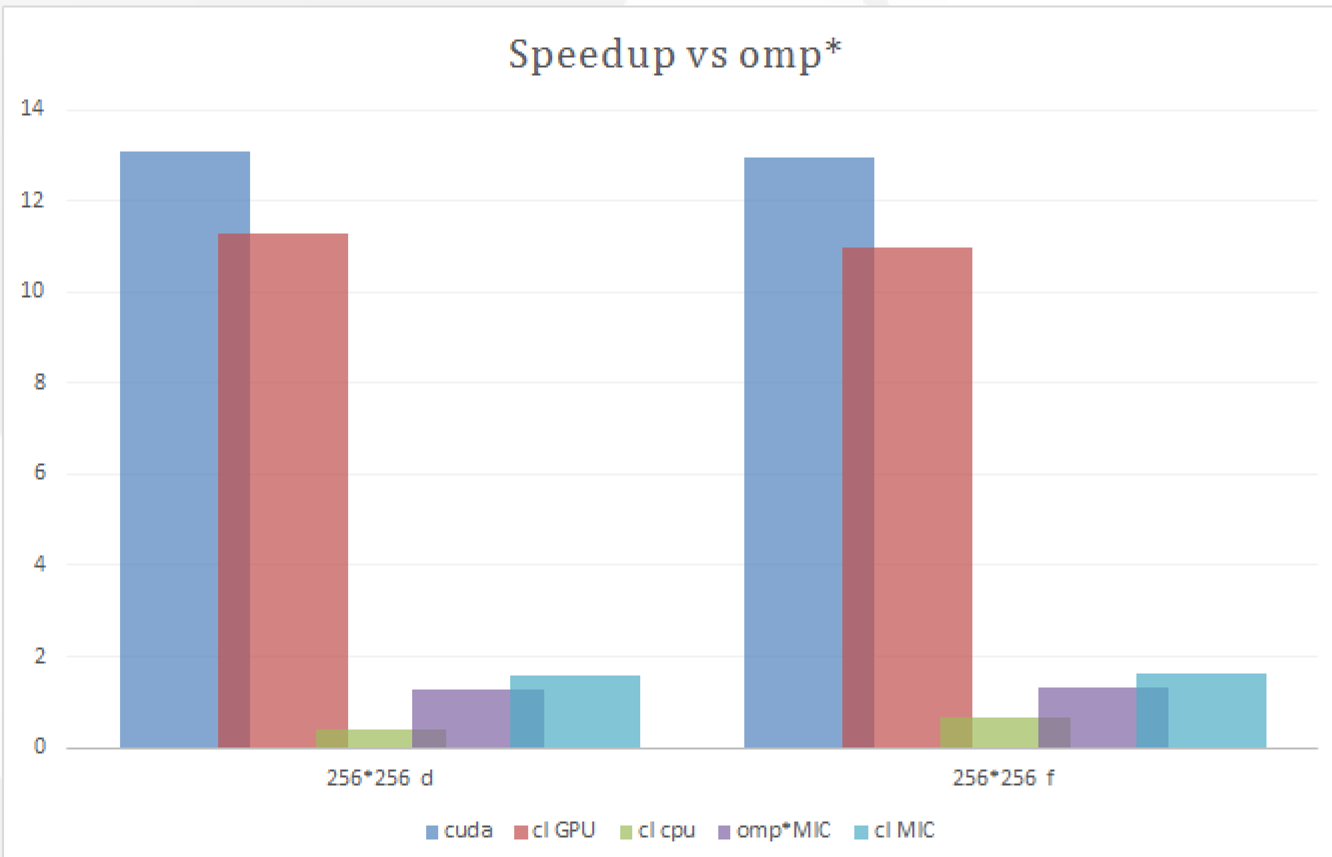
- Dot product Kernel Performance
  - *Baseline is 4-thread OpenMP w/ AVX



CPU: Intel(R) Core(TM) i5-2400, 4 cores @ 3.10GHz
MIC: Intel® Xeon Phi 7100 Series, 61 core @1.238 GHz.
GPU: Tesla K20c

MetaMorph: Performance vs. CPU OpenMP+AVX

- Transpose Kernel Performance
  - *Baseline is 4-thread OpenMP w/ AVX



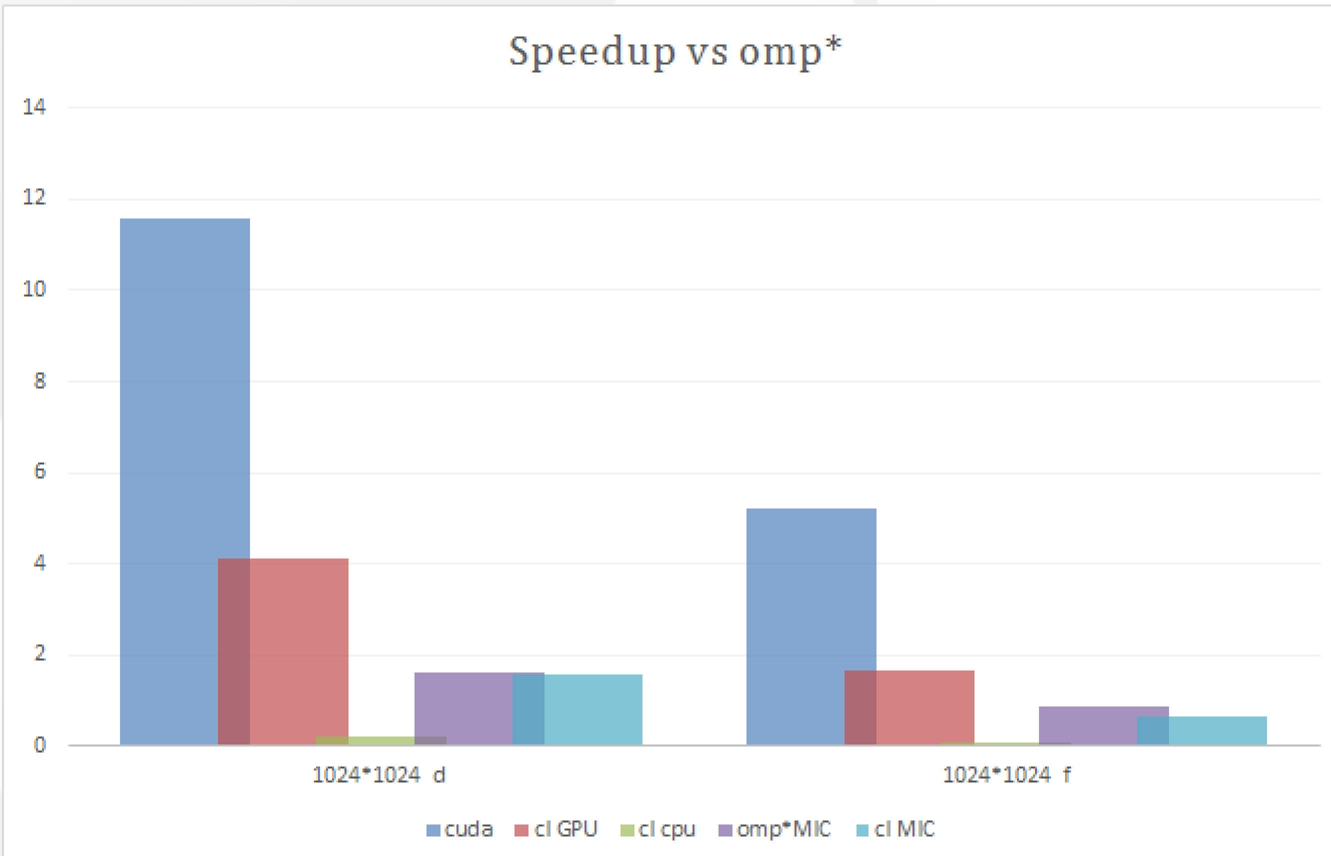
CPU: Intel(R) Core(TM) i5-2400, 4 cores @ 3.10GHz

MIC: Intel® Xeon Phi 7100 Series, 61 core @1.238 GHz.

GPU: Tesla K20c

MetaMorph: Performance vs. CPU OpenMP+AVX

- Data Marshaling Kernel Performance
  - *Baseline is 4-thread OpenMP w/ AVX



CPU: Intel(R) Core(TM) i5-2400, 4 cores @ 3.10GHz

MIC: Intel® Xeon Phi 7100 Series, 61 core @1.238 GHz.

GPU: Tesla K20c

MIC performance analysis/Optimization:

- Use single core Performance Degradation Factor (PDF) to analyze the performance on MIC and CPU

$$PDF_{MIC} = PDF_{FREQ} * PDF_{CMP} * PDF_{VEC} * PDF_{MEM}$$

  - PDF_{MIC} is the performance degradation factor on MIC in comparison with CPU (single core execution).
  - PDF_{FREQ} is the performance degradation factor due to frequency.
  - PDF_{CMP} is the performance degradation factor due to the core complexity.
  - PDF_{VEC} is the performance degradation factor due to the vectorization unit.
  - PDF_{MEM} is the performance degradation factor due to the memory system (working set).

MIC performance analysis/Optimization:

- Single core performance modeling
  - Dot product kernel
    - PDF_{MIC} ~= 3.5
      - PDF_{VEC} = 0.5 (2x speed up due to the wider vector unit)
      - PDF_{MEM} ~= 1 (There is no temporal locality, and the spatial locality depends on the cache line size)
  - Transpose kernel
    - PDF_{MIC} ~= 30
      - PDF_{VEC} = 1 (Memory transfer dominant)
      - PDF_{MEM} ~= 4 (CPUs outperform Intel MIC in workloads sensitive to locality/cache size)
  - Data marshaling
    - PDF_{MIC} ~= 12
      - PDF_{VEC} = 1 (Memory transfer dominant)
      - PDF_{MEM} ~= 1.5 (CPUs outperform Intel MIC in workloads sensitive to locality/cache size)

CPU: Intel(R) Core(TM) i5-2400, 4 cores @ 3.10GHz
MIC: Intel® Xeon Phi 7100 Series, 61 core @1.238 GHz.

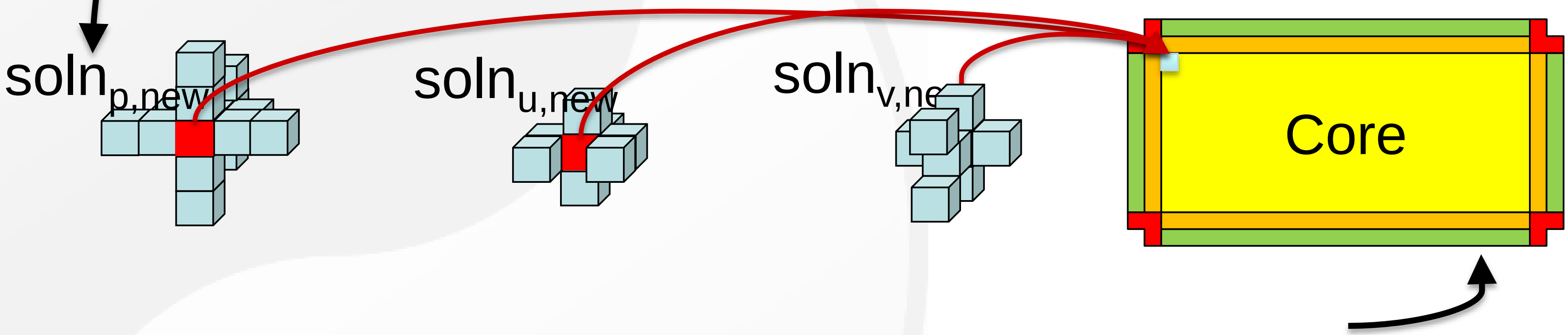
MIC performance analysis/Optimization:

- So our hypothesis: CPUs outperform Intel MIC in workloads sensitive to locality/cache size is correct.

- How we can use this to improve the performance?
  - CPU backend optimize for load balance vs. locality.
    - Dynamic scheduling with small data chunks.
  - MIC backend should optimize for locality over load balance
    - It is more sensitive to locality due to the lack of shared level3 cache to capture large working set.
    - Static scheduling with large (sequential) data chunks.
    - Collapsing nested loops to have wider workload distribution space (larger number of threads).

Work in progress

- Preliminary work towards stencil support
  - stencil shapes are exotic - library of standard shapes would be insufficient



    - (LDC) 9 conditional regions, but only 4 types of computation behavior
  - Plan: allow arbitrary stencils to be expressed as semi-structured C
    - Map ingested C to each platform via runtime modification of the kernel string
    - Utilize just-in-time (JIT) compilation for each platform

Future Work

- MPI plugin:
  - Automatic pipelined transfers (a la MPI-ACC)

- Adaptive runtime “meta-backend”
  - CoreTSAR /AfinityTSAR – support automatic scheduling across a complete node

- Domain decomposition plugin
  - Graph partitioning and load-balancing algorithms

- More kernels!
  - Start with those needed from the CFD space:
    - stencil computations, linear algebra, solvers, preconditioners
  - Expand with kernels from other domains
    - Computational Bio, Bigdata, Cosmology, etc.
  - Optimized for more devices
    - autotuning params, automatic device detection, JIT reconfig of OpenCL

Conclusion/Questions

- MetaMorph is designed to be a “build your own” library of computational primitives, accelerated from the start, instead of retroactively

- We want to enable domain scientists to write fast codes faster, and be able to maintain it themselves

- Plenty of work left to be done
  - Need more developers to help implement kernels and plugins, starting with CFD communication and branching from there

What else do I work on?

- Parallel circuit simulation:
  - **Helal, Ahmed E.**, Amr M. Bayoumi, and Yasser Y. Hanafy. "Parallel circuit simulation using the direct method on a heterogeneous cloud." Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE. IEEE, 2015.
  - Hassan, M. W. , **Helal, Ahmed E.**, Hanafy, Y. Y., "High Performance Sparse LU Solver FPGA Accelerator using a Static Synchronous Data Flow Model", The 23rd IEEE International Symposium on Field-Programmable Custom Computing Machines, FCCM'15, May 2015.

- AutoMatch:
  - A tool for automatic kernel-architecture matching
  - Enable scientists to predict the best hardware device for their applications without writing code for every target device.

References

- H. Jasak, et. al. , “OpenFOAM: A C++ library for complex physics simulations.” International Workshop on Coupled Methods in Numerical Dynamics, vol. 1000, pp. 1-20, 2007 http://www.openfoam.com/
- D. Lukarski, “PARALUTION project v0.7.0,” 2012. http://www.paralution.com/
- J. Dongarra, et. al. , “Accelerating Numerical Dense Linear Algebra Calculations with GPUs,” in Numerical Computations with GPUs. Springer, 2014, pp. 3–28.
- M. A. Heroux, et. al.“An overview of the Trilinos project,” ACM Trans. Math. Softw., vol. 31, no. 3, pp. 397–423, 2005.
- M. J. Harvey, et. al., “Swan: A tool for porting CUDA programs to OpenCL,” Computer Physics Communications, vol. 182, no. 4, pp. 1093–1099, 2011.