MetaMorph
A Modular Library of Malleable Accelerator Primitives for Heterogeneous Parallel Computing

SEEC Fall Symposium 2015
Paul Sathre, Ahmed Helal
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)

```
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];
}
```
Motivation:
Productivity = Programmability + Performance + Portability

• Dot Product (Parallel on GPU using CUDA)

HOST

```c
1. int main(int argc, char ** argv)
2. {
3.     int n_elem;
4.     double *a, *b, *ret, sum = 0.0;
5.     //allocate and initialize a, b, and ret
6.     ... 
7.     double *dev_a, *dev_b, *dev_r;
8.     size_t size = n_elem * sizeof(double);
9.     //allocate device buffers
10.    cudaMalloc((void**)&dev_a, size);
11.    cudaMalloc((void**)&dev_b, size);
12.    cudaMalloc((void**)&dev_r, size);

13.    //initialize device buffers
14.    cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
15.    cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

16.    //set grid/block size
17.    ...
18.    //multiply elements
19.    dotProd-vmul<<<grid, block>>>(dev_a, dev_b, dev_r, 
20.        n_elem);

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

23.    //CPU sum
24.    int i;
25.    for (i = 0; i < n_elem; i++)
26.        sum += ret[i];
27. }
```

DEVICE (GPU)

```c
1. __global void dotProd-vmul(
2.     double *a, double *b,
3.     double *ret, int n_elem)
4. {
5.     int i;
6.     int tid = blockIdx.x * 
7.         blockDim.x + threadIdx.x;
8.     int n_threads = blockDim.x * 
9.         gridDim.x;
10.    for (i=tid; i<n_elem; i+=n_threads)
11.        ret[tid] = a[tid] * b[tid];
12. }
```

```c
1. __kernel void dotProd-vmul(
2.     double *a, double *b,
3.     double *ret, int n_elem)
4. {
5.     int i;
6.     int tid = blockIdx.x * 
7.         blockDim.x + threadIdx.x;
8.     int n_threads = blockDim.x * 
9.         gridDim.x;
10.    for (i=tid; i<n_elem; i+=n_threads)
11.        ret[tid] = a[tid] * b[tid];
12. }
```
Motivation:
Productivity = Programmability + Performance + Portability

Dot Product (Optimized Parallel on GPU Device)

HOST

1. int argc = argv[0]; // arguments
2. { //global and block sizes
3. int ni, nj, nk; // ty, tr;
4. // pick i, j and block device
5. cudaSetDevice(0);
6. //declare host memory
7. double *a, *b, *ret, zero = 0.0;
8. __global__ void kernel_dotProd(double *phi1, double *phi2, int i, int j, int k, int sx);
9. int sy, sz, ex, ey, itr;
10. bool boundx, boundy, boundz;
11. size_t size = sizeof(double) * ni * nj * nk;
12. double *dev_a, *dev_b, *dev_r;
13. cudaMalloc(&dev_a, size);
14. cudaMalloc(&dev_b, size);
15. cudaMalloc(&dev_r, sizeof(double));
16. //initialize them
17. cudaMemcpyp(dev_a, a, size,
18. cudaMemcpyHostToDevice);
19. cudaMemcpyp(dev_b, b, size,
20. cudaMemcpyHostToDevice);
21. cudaMemcpyp(dev_r, &zero, sizeof(double),
22. cudaMemcpyHostToDevice);
23. //set computation shape
24. grid = dim3((ni+tx-1)/ni, (nj+ty-1)/ty, 1);
25. blockDim.x = (ki+tx)/ki;
26. blockDim.y = (kj+ty)/kj;
27. blockDim.z = (kn+tz)/kn;
28. start = dim3(0, 0, 0);
29. end = dim3(ni-1, nj-1, nk-1);
30. //run the kernel
31. dotProd<<<grid, block, shape, start, end>>>
32. (dev_a, dev_b, shape.x, shape.y, shape.z,
33. start.x, start.y, start.z, end.x, end.y, end.z,
34. (nk+tz-1)/tz, dev_r, tx*ty*tz);
35. //bring the dot product back
36. cudaMemcpyp(dev_r, ret, sizeof(double),
37. cudaMemcpyHostToDevice);
38. return 0;
39. }
40. //Implementation of double atomicAdd
41.
42. __device__ void block_reduction(double *psum,
43. int tid, int len_)
44. { //allocate and initialize it
45. start = dim3(0, 0, 0);
46. cudaMalloc(&psum[tid], len);
47. cudaMemcpyHostToDevice
48. psum[tid] = 0;
49. //set computation shape
50. grid = dim3((ni+tx-1)/ni, (nj+ty-1)/ty, 1);
51. blockDim.x = (ki+tx)/ki;
52. blockDim.y = (kj+ty)/kj;
53. blockDim.z = (kn+tz)/kn;
54. start = dim3(0, 0, 0);
55. end = dim3(ni-1, nj-1, nk-1);
56. //run the kernel
57. dotProd<<<grid, block, shape, start, end>>>
58. (dev_a, dev_b, shape.x, shape.y, shape.z,
59. start.x, start.y, start.z, end.x, end.y, end.z,
60. (nk+tz-1)/tz, dev_r, tx*ty*tz);
61. //bring the dot product back
62. cudaMemcpyp(dev_r, (ret, sizeof(double),
63. cudaMemcpyHostToDevice);
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)

```c
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;
    grid[0] = (ni+tx-1)/ni, grid[1] = (nj+ty-1)/ty,
    grid[2] = (nk+tz-1)/tz;
    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

Virginia Tech
Invent the Future

Synergy.cs.vt.edu
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

<table>
<thead>
<tr>
<th></th>
<th>MetaMorph</th>
<th>OpenFOAM</th>
<th>Paralution</th>
<th>MAGMA</th>
<th>Trilinos</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFD support</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>Acceleration</td>
<td>OMP, CUDA, OCL</td>
<td>No internal support</td>
<td>OMP, CUDA, OCL</td>
<td>OMP, CUDA, OCL</td>
<td>preliminary CUDA/MIC</td>
</tr>
<tr>
<td>support</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Target devices</td>
<td>CPU, GPU, MIC</td>
<td>CPU</td>
<td>CPU, GPU, MIC</td>
<td>CPU, GPU, MIC</td>
<td>CPU, GPU, MIC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI &amp;</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>interoperability</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>framework-centric development</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>External</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>dependencies</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

User Apps
Purpose-built to provide single API to accelerated kernels for current and future devices
Runtime control over backends, plugins, and accelerator device selection

C API
Timer Plugin
MPI Plugin
Fortran 2003 API
Future Plugins

Future Device Backends

CUDA Backend
OpenCL Backend
OpenMP Backend

-D WITH_CUDA
-D WITH_OPENCL
-D WITH_OPENMP

-D WITH_TIMERS
-D WITH_MPI
-D WITH_FORTRAN
-D WITH_PLUGINX

Backend Layer
Plugin Layer
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

Time per element (μs)

Vector Length

PLASMA/MKL float (CPU)
PLASMA/MKL double (CPU)
ATLAS float (CPU)
ATLAS double (CPU)
MetaMorph+OpenCL float (GPU)
MetaMorph+OpenCL double (GPU)
MetaMorph+CUDA float (GPU)
MetaMorph+CUDA double (GPU)
clAmdBlas float (GPU)
clAmdBlas double (GPU)
cuMAGMA float (GPU)
cuMAGMA double (GPU)
MetaMorph: Transpose performance

Time per element (μs)

Transpose Size

- MKL float (CPU)
- MKL double (CPU)
- MetaMorph+OpenCL float (GPU)
- MetaMorph+OpenCL double (GPU)
- MetaMorph+CUDA float (GPU)
- MetaMorph+CUDA double (GPU)
- cuMAGMA float (GPU)
- cuMAGMA double (GPU)

Virginia Tech
Invent the Future
synergy.cs.vt.edu
MetaMorph: Library overhead vs. standalone kernels

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

Execution Time (us)

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
MetaMorph: Performance vs. CPU OpenMP+AVX

- Transpose Kernel Performance
  - *Baseline is 4-thread OpenMP w/ AVX
MetaMorph: Performance vs. CPU OpenMP+AVX

- Data Marshaling Kernel Performance
  - *Baseline is 4-thread OpenMP w/ AVX
MIC performance analysis/Optimization:

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

\[ \text{PDF}_{\text{MIC}} = \text{PDF}_{\text{FREQ}} \times \text{PDF}_{\text{CMP}} \times \text{PDF}_{\text{VEC}} \times \text{PDF}_{\text{MEM}} \]

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

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

• 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


• D. Lukarski, “PARALUTION project v0.7.0,” 2012. http://www.paralution.com/

