Raising the Level of Many-core Programming with Compiler Technology

meeting a grand challenge

Wen-mei Hwu
University of Illinois, Urbana-Champaign
Agenda

• Many-core Usage Pattern and Workflow
• Using Kernels in High-level Environments
• Available Kernels
• Raising the Level of Kernel Development
• Conclusion and Outlook
GPU computing is catching on.

- 280 submissions to GPU Computing Gems
  - 110 articles included in two volumes
GPU computing examples in visual applications

Real-time emotion analysis

Real-time surface reconstruction

Real-time gesture analysis
A Common GPU Usage Pattern

• A desirable approach considered impractical
  – Due to excessive computational requirement
  – But demonstrated to achieve domain benefit
  – Convolution filtering (e.g. bilateral Gaussian filters), De Novo gene assembly, etc.

• Use GPUs to accelerate the most time-consuming aspects of the approach
  – Kernels in CUDA or OpenCL
  – Refactor host code to better support kernels

• Rethink the domain problem
Problem Formulation

Find or create library kernels

Debug, Test, Enough Speed?

Identify performance critical kernels

Fine-grain parallelization with refactoring

Debug, Test, Enough Speed?

Coarse-grain parallelization with refactoring

Debug, Test, Enough Speed?

Done
USING KERNELS IN HIGH-LEVEL ENVIRONMENTS
CUDA/OpenCL Kernel Interface

```c
__global__ void v_add(float *c, float *a, float *b, size_t size)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if(idx < size) c[idx] = a[idx] + b[idx];
}

int main(int argc, char *argv[])
{
    float *h_a, *h_b, *h_c, *d_a, *d_b, *d_c;
    size_t size = LENGTH * sizeof(float);
    assert((h_a = malloc(size) != NULL);
    assert((h_b = malloc(size) != NULL);
    assert((h_c = malloc(size) != NULL);
    assert(cudaMalloc((void **)&d_a, size) == cudaSuccess));
    assert(cudaMalloc((void **)&d_b, size) == cudaSuccess));
    assert(cudaMalloc((void **)&d_c, size) == cudaSuccess));
    read_file(argv[1], h_a);
    read_file(argv[2], h_b);
    assert(cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice) == cudaSuccess);
    assert(cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice) == cudaSuccess);
    ...;
    v_add<<<Dg, Db>>>(d_c, d_a, d_b, LENGTH);
    assert(cudaThreadSynchronize() == cudaSuccess);
}
```
Need for Better Kernel Interface

• Tedious and error prone
  – Each object has two versions
  – Programmers try to avoid copying by tracking which object is needed by CPU and GPU at each program point
  – Double buffering and triple buffering in copying used to hide latency

• In real applications
  – Disk I/O and MPI messages are often done by different threads than compute to hide latency
  – Data copying involves tricky thread synchronization
Thrust – C++ STL for CUDA

• Containers
  – Make common operations concise
  – Hide cudaMalloc(), cudaMemcpy(), cudaFree()
  – Vector, list, map, …

• Iterators
  – Pointers to the beginning and end of containers

• Algorithms
  – Commonly used data parallel kernels

Thrust Use Example

// initialize random values on host
thrust::host_vector<int> h_vec(1000);
thrust::generate(h_vec.begin(), h_vec.end(), rand);

// copy values to device
thrust::device_vector<int> d_vec = h_vec;

// compute sum on host
int h_sum = thrust::reduce(h_vec.begin(), h_vec.end());

// compute sum on device
int d_sum = thrust::reduce(d_vec.begin(), d_vec.end());
MatLab, Python, and more

• Jacket- use of CUDA functions in MatLab code
  – Automated GPU memory management and data transfer
  – Retains MatLab interpreter execution model
  – http://wiki.accelereyes.com

• PyCUDA- use of CUDA functions in Python code
  – Consistent with numpy usage
  – http://mathema.tician.de/software/pycuda

• GMAC – automatic management of GPU kernel memory and data transfer.
  – http://adsm.googlecode.com/
Advanced Compiler Technology?

• Currently, these systems do not use advanced compiler technology or hardware support
  – Implemented mostly through templates/macros

• Unnecessary data transfers before and after kernel execution
  – Could be optimized away with information on side-effect of kernels and live range of data objects
  – Portability for fusion architectures

• Double and triple buffering done by hand
  – Could be automated with compiler transformations
AVAILABLE KERNELS
CUBLAS

- Basic Linear Algebra Subprograms
  - single & double precision, real & complex values
    - Level 1: vector-vector (e.g., SAXPY)
    - Level 2: matrix-vector (e.g., DGEMV)
    - Level 3: matrix-matrix (e.g., DGEMM)

- CUBLAS implements standard BLAS interface
  - Column major layout
  - High data transfer overhead

- Room for performance improvements for less common matrix shapes (e.g., Jalby)
CULA: LAPACK

Also, see Magma - http://icl.cs.utk.edu/magma
CUSP – Sparse Methods

• Generic data structure and algorithms
  – For sparse matrices and graphs
  – Several sparse matrix and graph storage formats

• Supports high-level usage
  – C++ templated BLAS wrappers
  – High-performance SpMV kernel library
  – Simple Conjugate Gradient (CG) solvers and Biconjugate Gradients Stabilized Methods (BiCGSTAB)
  – Need direct solvers, eigenanalysis, etc.
Other Libraries

- **Graph Algorithms**
  - BFS routines exist, CUSP covers some sparse graph algorithms
  - Need graph partitioning tools (e.g., METIS)

- **Unstructured Grid Algorithms**
  - Some 3D surface mesh generation and refinement routines exist (as used in graphics)
  - Need 3D volume mesh generation (e.g., CGAL) and more advanced refinement tools
Other Libraries (cont.)

• Spatial Data Structures
  – Used in input data binning
  – Need Oct-trees, KD-trees, BSP-trees, and uniform-grid spatial sorting

• Hidden Markov Models
  – Used in speech recognition, gene sequencing, character recognition, etc.
  – Need training, testing, analysis tools (e.g., HKT)

• Add your own favorite libraries here…
Kernel development for GPUs is heavy lifting.

Each kernel is typically a 3-month job but very few developers benefit from advanced compiler technology today.

Little code reuse due to kernel sensitivity to memory access patterns and work distribution.
Many-core Kernel Development

• Many-core programming is about performance and scalability.
  – Scalability is also key to power efficiency.
  – Performance and scalability for many-cores requires largely the same techniques.
    • To regularize work and data for massively parallel execution.
    • To localize data for conserving memory bandwidth

• There is a gap between what the programmers need and what the tools provide today.
Key to Massive Parallelism - Regularity and Locality
Phillip Colella’s “Seven dwarfs’ High-end simulation in the physical sciences = 7 numerical methods:

1. Structured Grids (including locally structured grids, e.g. Adaptive Mesh Refinement)
2. Unstructured Grids
3. Fast Fourier Transform
4. Dense Linear Algebra
5. Sparse Linear Algebra
6. Particles
7. Monte Carlo

- If add 4 for embedded, covers all 41 EEMBC benchmarks
  8. Search/Sort
  9. Filter
  10. Combinational logic
  11. Finite State Machine

Well-defined targets from algorithmic, software, and architecture standpoint

Slide from “Defining Software Requirements for Scientific Computing”, Phillip Colella, 2004
Seven Techniques in Many-core Programming (so far)

1. Scatter to Gather transformation
2. Granularity coarsening and register data reuse
3. Data access tiling
4. Data layout and traversal ordering
5. Input Data Binning
6. Bin sorting and partitioning for non-uniform data
7. Hierarchical queues/kernels for dynamic data

http://courses.engr.illinois.edu/ece598/hk/
GPU Computing Gems, Vol. 1 and 2
1: Scatter to Gather Transformation
2. Granularity Coarsening and Register Data Reuse

- Parallel execution often requires redundant and coordination work
  - Merging multiple threads into one allows reuse of result, avoiding redundant work
3: Data Access Tiling
4. Data Layout Transformation

Array of Structure: [z][y][x][e]

Structure of Array: [e][z][y][x]
4X faster than AoS on GTX280

[2][y_{31:4}][x_{31:4}][e][y_{3:0}][x_{3:0}], 6.6 X faster than AoS
5: Input Data Binning

Bins far beyond the cutoff distance are never scanned.
6. Bin Sorting and Partitioning

Variable sized bins, sort and scan

On-chip Memory

On-Chip Memory

CPU

Variable sized bins, sort and scan

On-chip Memory

On-Chip Memory
7. Hierarchical Queues and Kernels

w-queue
b-queue
On-chip Memory

Work Threads
Dummy Threads

Propagate

Level i
Level i+1
Level i+2

PACT 2010
Tools go with techniques.

• Tools should facilitate key techniques
  – Programmers should write code “for others to understand instead of for computers to execute”
    - Dijkstra

• Techniques vary in their potential for automation
  – Scatter-to-gather, granularity coarsening, data access tiling, and memory layout quite amenable
    • Need clear performance guidance
  – Input binning, bin sorting, and hierarchical queues are much harder
    • Need to provide APIs understood by compilers/tools
    • Developer feedback critical to success
Some Current Efforts

• PGI FORTRAN (PGI)
  – FORTAN loop compilation into kernels

• Chapel (CRAY)
  – Chapel vector loop compilation into kernels

• Copperhead (Berkeley)
  – Annotated Python subset JITed into kernels
  – Modest performance within interpreter use model

• Pyon (Illinois)
  – Strongly typed Python subset compilation into kernels
  – Aggressive memory transformations
ADAPT: Example of Advanced Compiler Technique in kernel performance prediction

HW constraints enable efficient abstract interpretation to emulate expert-level performance prediction

Baghsorkhi and Hwu, EPHAM 2009, PPoPP 2010
Conclusion and Outlook

• Standard and domain library kernels are main use model of many-cores.
  – Kernel development is heavily lifting
  – Useful abstractions will require compilers to do some heavy lifting.

• However, compilers/tools are fragile.
  – Compilers transformations need to be part of the development, rather than afterthought
  – Developers must be able to reach into the abstraction whenever they want.
Crossing the Valley of Death

By giving developers what they want.
Acknowledgements


• And many others!
THANK YOU!
OpenCurrent – PDE Solvers

• Structured Grids
  – 1D, 2D, 3D - Single and double precision
  – Linear combinations, host-device transfers, interpolation at non-grid points, array-wide reduction

• Solvers
  – Calculate terms from discretization of PDEs
  – Finite-difference advection and diffusion schemes
  – Multi-grid solver for Poisson Equations

• Equations
  – Solve time-dependent PDEs
  – Incompressible Navier-Stokes solver, …
  – Need many more types of PDEs