Compiler-Based Autotuning Technology

Lecture 4: Parallel Code Generation and CUDA-CHiLL

Mary Hall

July, 2011

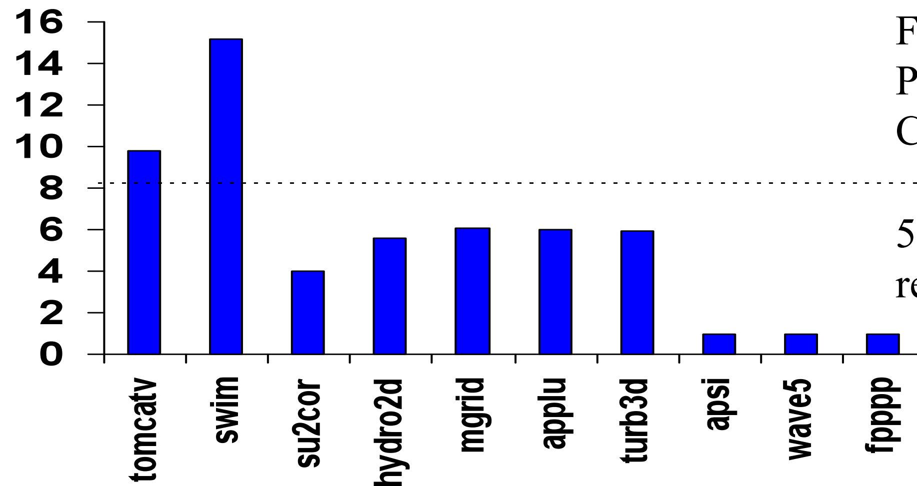
* This work has been partially sponsored by DOE SciDAC as part of the Performance Engineering Research Institute (PERI), DOE Office of Science, the National Science Foundation, DARPA and Intel Corporation.

Motivation

- What about parallel code?
  - Emerging multi-core and many-core architectures
  - New (CUDA and OpenCL) and old (OpenMP data-parallel programming models
  - Heterogeneous systems
- Different CHiLL scripts can describe mapping for different architectures, applied to the same input code
  - Portable, heterogeneous support
  - Today we'll generate CUDA code

THE UNIVERSITY OF UTAH

Previous Work in Automatic Parallelization



From Hall et al, "Maximizing Multiprocessor Performance with the SUIF Compiler", IEEE Computer, Dec. 1996.

50% higher Specfp95 ratio than previously reported

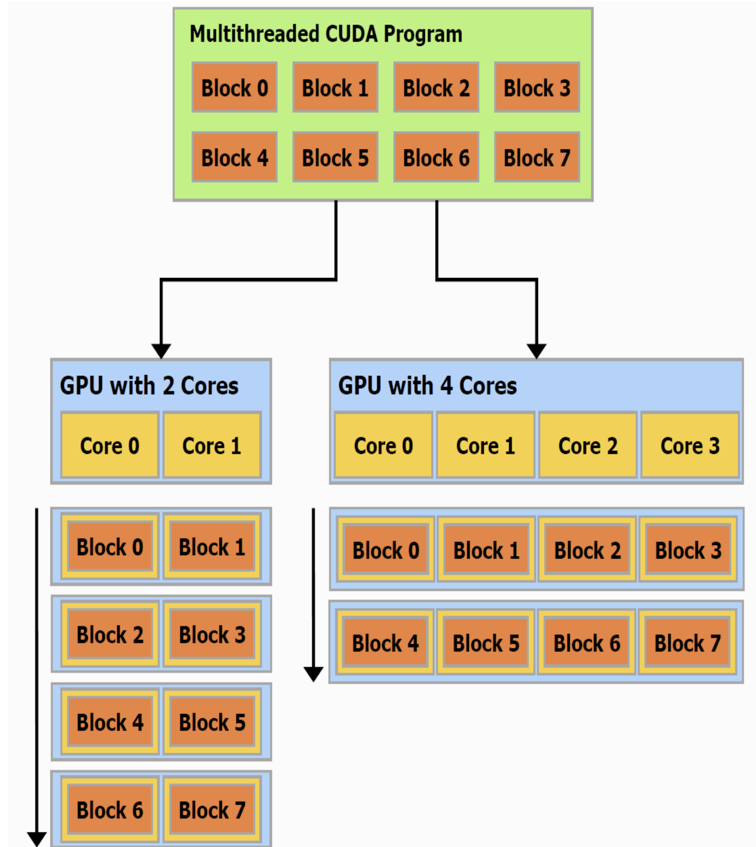
- Old approaches:
  - Limited to loops and array computations
  - Difficult to find sufficient granularity (parallel work between synchronization)
  - Success but from fragile, complex software
- New ideas:
  - Finer granularity of parallelism (more plentiful)
  - Combine with hardware support (e.g., speculation and multithreading)
  - Input from the user and autotuning

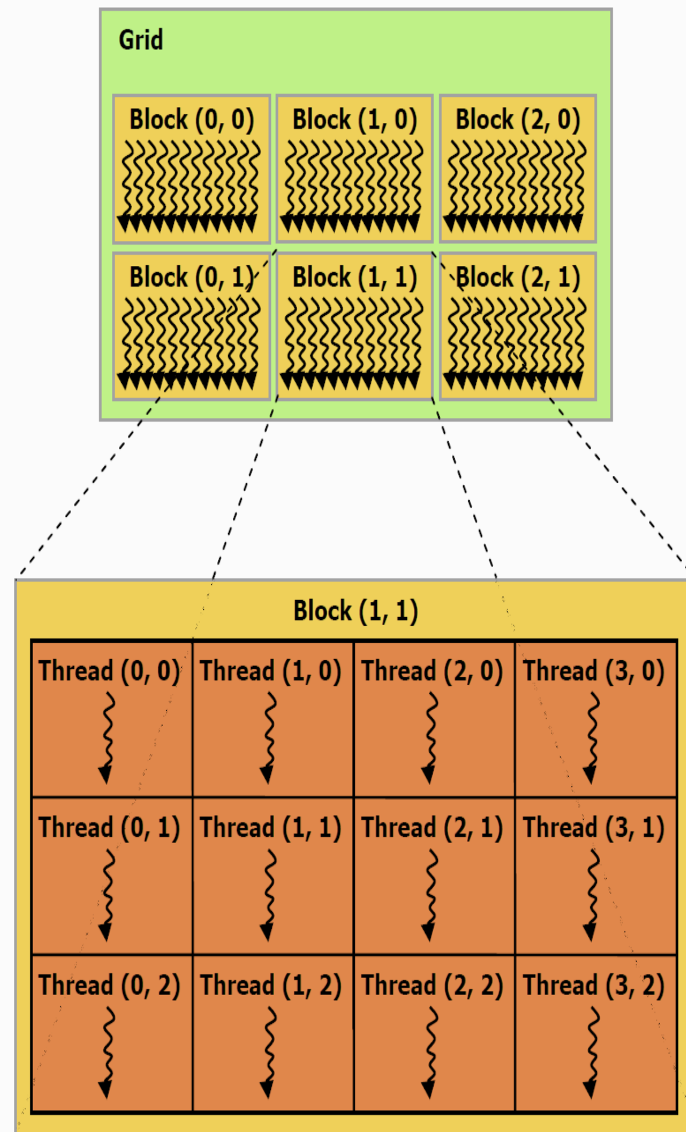
THE UNIVERSITY OF UTAH

Outline for Today's Lecture

1. Basics on GPUs and GPU code generation
2. A programming language interface adds another layer of abstraction
3. CUDA-CHiLL, Automatic Parallelization for GPUs
   a. Example, Matrix-Vector Multiply
   b. Transformation Strategy Generator
   c. Example, Matrix-Matrix Multiply for two GPU architectures
   d. Other examples: Convolution and MRI-Q
4. Productivity improvements
5. Extensive performance results

1. Nvidia GPU Basics: Two-Level Parallelism Hierarchy





© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE498AL, University of Illinois, Urbana-Champaign

1. Nvidia GPU Basics: Complex GPU Memory Hierarchy

Global Memory

–1GB for GTX 280

–Access 100-200 cycles

–Bandwidth optimization:

* Coalesced global memory accesses
* Neighboring threads access adjacent data

–Also, Texture/Constant Memory

Shared Memory

–16K per SM

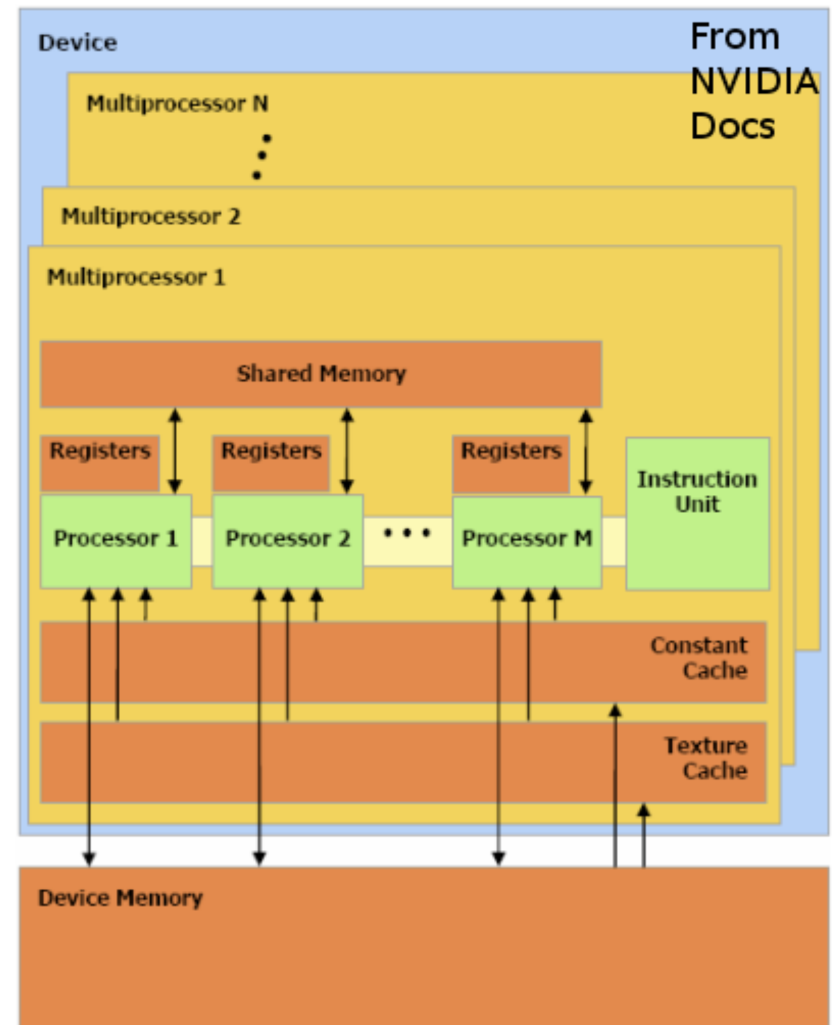
–Shared across threads of SM

Register File

–16K per SM

–Thread local data

Data Cache on Fermi only



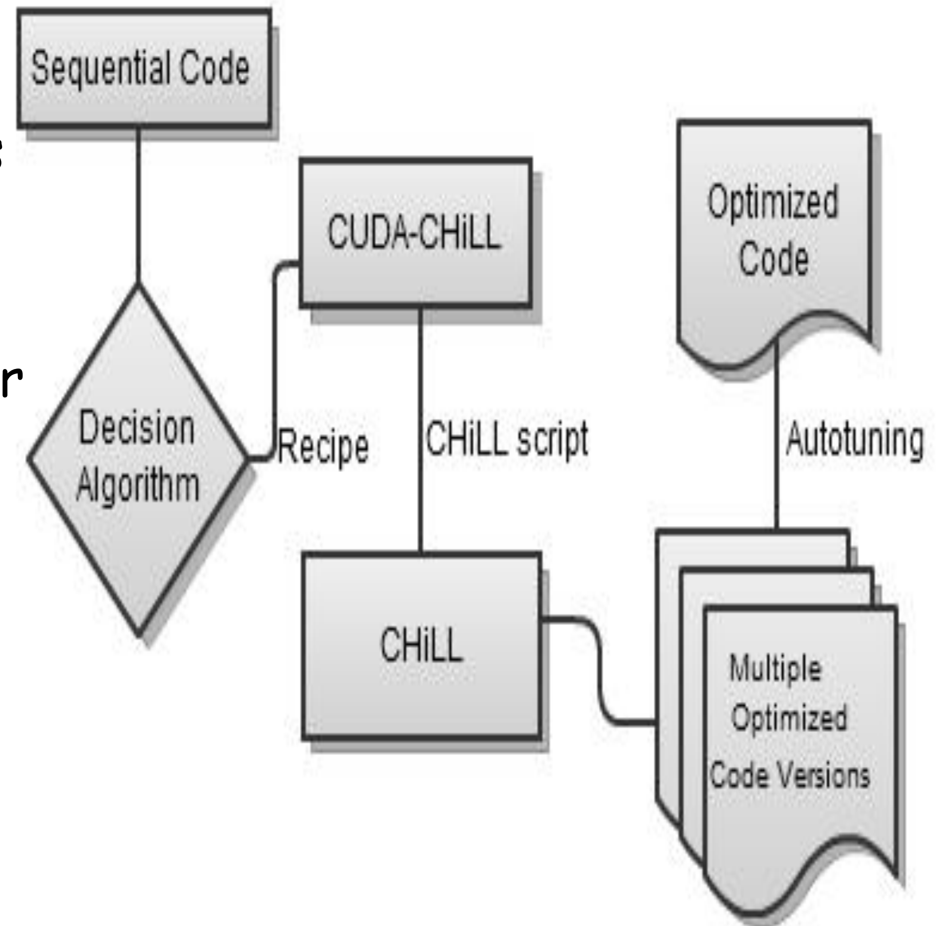
© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE498AL, University of Illinois, Urbana-Champaign

1. GPU Code Generation Requirements

- Identify two levels of parallelism
  - Block-level parallelism must use very expensive synchronization to protect global memory (so should be avoided)
  - Barrier synchronization available at the thread level
- Data placement in the very complex memory hierarchy
  - Registers for thread-local data
  - "Shared" (scratchpad) memory for data shared across threads in a block
  - Global memory for data with no reuse
  - Texture and constant memory for read-only, reused data
- Optimizations to improve memory bandwidth
  - Coalesce global memory accesses
  - Avoid shared memory bank conflicts
  - Parallel access using texture memory

2. CUDA-CHiLL System

- Input: Sequential C Code
- Transformation Strategy Generator (TSG) automatically generates transformation recipes
  - Computational decomposition
  - Data Staging
- High-level CUDA abstraction layer generates CHiLL primitives
  - Compact scripting language
- CHiLL primitives
  - Transformation and code generation
- Autotuning and pruning of the search space



THE UNIVERSITY OF UTAH

2. Higher-Level Abstraction in CUDA-CHiLL System

- High-level, programmable interface to transformation and code generation – a ***programming language*** approach
  - Encapsulation and control flow
  - Queries to compiler guide optimization
  - Support users with different skill levels
- GPU Code Generation – ***CUDA-CHiLL***
  - Rapid compiler prototyping through scripting language (Lua)
  - Compact: CUDA-CHiLL is roughly 300 lines of Lua code
- A model for other context-specific abstraction using CHiLL

2. Recall CHiLL Set of Transformations

Transformation and Parameters	Description
permute ([stmt],[level],order)	Permute optional [stmt] to optional loop [level] according to order. Can omit [stmt] and [level] and entire loop nest is permuted.
unroll (stmt,level,unrollfactor)	Unroll loop at level for the subloop specified by stmt/level. Unroll by unrollfactor.
tile (stmt,level,ts,[outerlooplevel])	Tile loop at level for the subloop specified by stmt/level and tile size ts. Place controlling loop at optional [outerlooplevel] or defaults to outermost.
datacopy (stmt,level,array,[index])	Calculate footprint for all references to array in subloop specified by stmt/level and copy into temporary, replacing original accesses with copy. Optional [index] refers to fastest-changing dimension.
split(stmt,level,condition)	Split iteration space at subloop specified by stmt/level according to condition and its complement.
datacopy_privatized (stmt,level,array,[index])	Similar to datacopy, but creates a private copy in parallel thread code.
Other transformations include:	fuse, distribute, skew, scale, reverse, shift, peel, nonsingular

2. CUDA-CHiLL Set of Transformations

Command	Example Parameter	Description
tile_by_index	{“i”,”j”}	Indices of the loops to be tiled
	{TI,TJ}	Tile sizes for each index variable
	{l1_control=“ii”, l2_control= “jj”}	Tile controlling loop names
	{“ii”,”jj”,”I”,”j”}	Final loop order
cudaize	“gpuMV”	Name of generated kernel
	{a=N,b=N,c=N*N}	Sizes of input arrays
	{block={“ii”}, thread={“jj”}}	Indices of blocks and threads
copy_to_registers	“kk”	Loop level for copy
	“c”	Array to be copied
copy_to_shared	“tx”	Loop level for copy
	“b”	Array to be copied
	-16	Input to padding
Others:	copy_to_texture, copy_to_constant	
unroll_to_level	1	Level of loop nest to unroll

2. CUDA-CHiLL Tiling Algorithm

```
TileByIndexCommands(s,I,S,M,O)
Input: s: Statement number; I: Indices to tile; S: Tile sizes;
       M: Map of names for indices; O: Final loop nest order
Output: F: Set of transformation operations
begin
  F := ∅
  C := extract control loop name list from M
  I' := extract renamed tile loop name map from M ∪ I
  order := BuildOrder(O, C, I', 0)
  F := F + [permute(s,order)]
  for i in 1..|I| do
    level := FindLevel(I_i)
    order := BuildOrder(O, C, I', i)
    offset := offset between I'_i and C_i in order
    if offset < 0 then
      F := F + [tile(s,level,S_i,level + offset,I'_i,C_i)]
    then
      F := F + [tile(s,level,S_i,level,I'_i,C_i)]
    end
    order := BuildOrder(O, C, I',i)
    F := F + [permute(s,order)]
  end
  return F
```

```
tile_by_index(
  {"i","j"}, {TI,TJ},
  {l1_control="ii",
   l2_control="jj"},
  {"ii", "jj", "i", "j"})
```

```
permute(0, {"i", "j"})
tile(0, 1, 64, 1, "i", "ii", 1)
permute(0, {"ii", "i", "j"})
tile(0, 3, 16, 2, "j", "jj", 1)
permute(0, {"ii", "jj", "i", "j"})
```


THE UNIVERSITY OF UTAH

3a. Example: Matrix-Vector Multiply

```
for (i=0; i<N; i++)
   for (j=0; j<N; j++)
       a[i] = a[i] + c[j][i]*b[j];
```


THE UNIVERSITY OF UTAH

3a. TSG: Computation Decomposition for Matrix-Vector Multiply

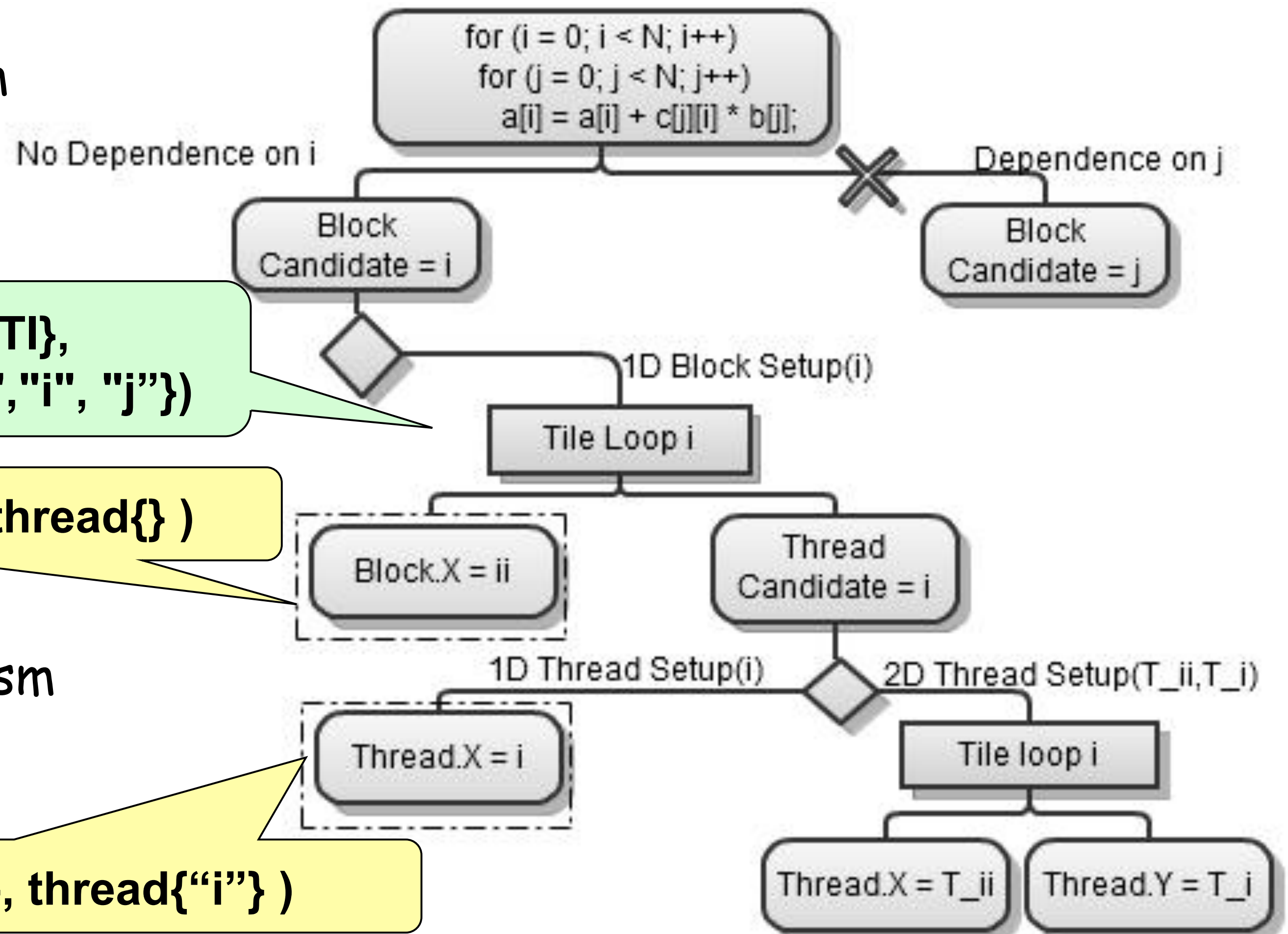
- Block Parallelism

tile_by_index({“i”}, {TI}, {l1_control="ii”},{"ii","i", "j”})

cudaize(block{“ii”}, thread{})

- Thread Parallelism

cudaize(block{“ii”}, thread{“i”})



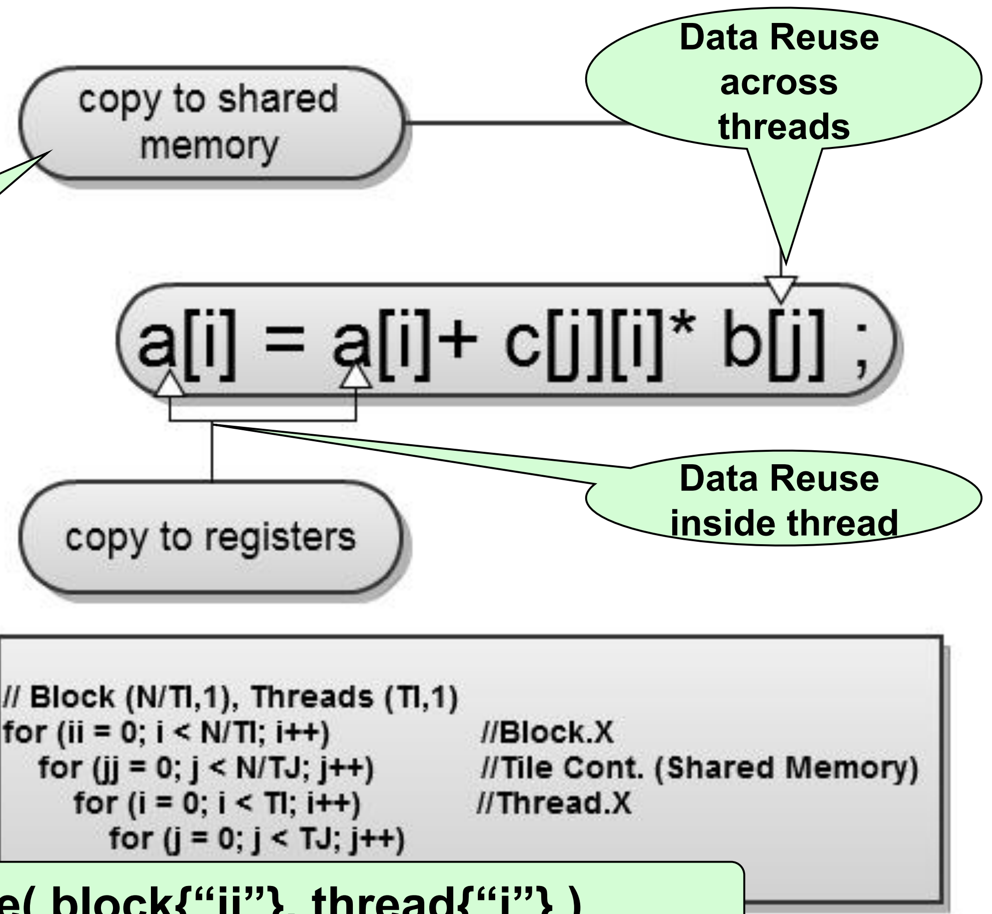
THE UNIVERSITY OF UTAH

3a. TSG: Data Staging for Matrix-Vector Multiply

- Data Staging
  - Shared Memory

tile_by_index({"j"}, {TJ}, {l1_control="jj"},{"ii","jj","i", "j"})

- Registers



Final Loop Order

- Cudaize
- Unrolling

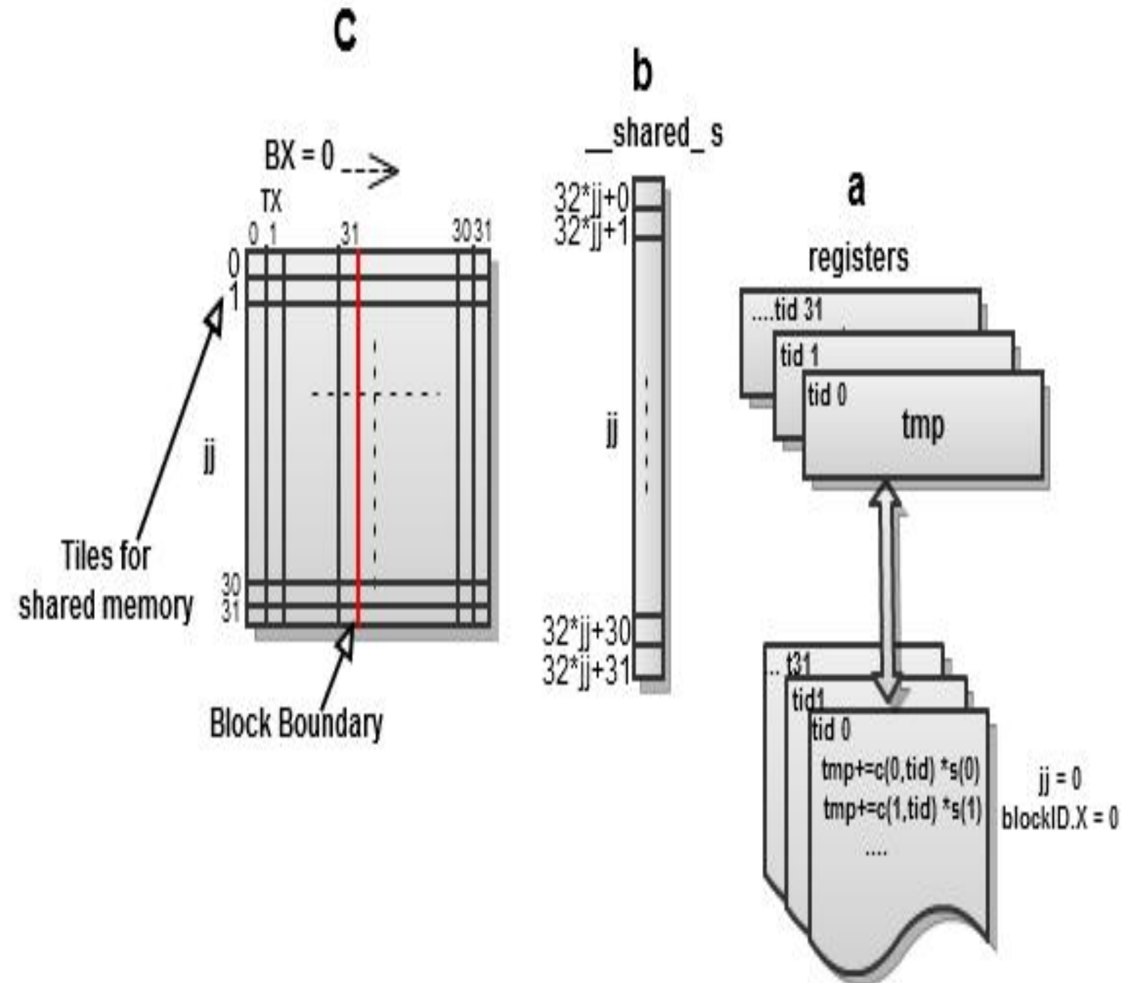
cudaize(block{"ii"}, thread{"i"})

unroll to depth(1)

THE UNIVERSITY OF UTAH

CUDA-CHiLL Recipe

```
N = 1024
TI= TJ = 32
tile_by_index({“i”,”j”},
    {TI,TJ},{l1_control="ii”,
    l2_control=“k”},{"ii",
    “jj”,"i", "j”})
normalize_index(“i”)

cudaize(“mv_GPU”, {a=N,
    b=N, c=N*N},{block=
    {“ii”}, thread={“i”}})

copy_to_shared(“tx”, “b”, 1)
copy_to_registers(“jj”, “a”)
unroll_to_depth(1)
```



THE UNIVERSITY OF UTAH

3a. Matrix-Vector Multiply: GPU Code

Generated Code: with Computational decomposition only.

```
__global__ GPU_MV(float* a, float* b, float** c) {
int bx = blockIdx.x; int tx = threadIdx.x;
int i = 32*bx+tx;
for (j = 0; j < N; j++)
                a[i] = a[i] + c[j][i] * b[j];
}
```

Final MV Generated Code: with Data staged in shared memory & registers.

```
__global__ GPU_MV(float* a, float* b, float** c) {
int bx = blockIdx.x; int tx = threadIdx.x;
__shared__ float bcpy[32];
float acpy = a[tx + 32 * bx];
for (jj = 0; jj < 32; jj++) {
    bcpy[tx] = b[32 * jj + tx];
    __syncthreads();
   //this loop is actually fully unrolled
   for (j = 32 * jj; j <= 32 * jj + 32; j++)
      acpy = acpy + c[j][32 * bx + tx] * bcpy [j];
     __syncthreads();
}
a[tx + 32 * bx] = acpy;
}
```


THE UNIVERSITY OF UTAH

3b. TSG: 3-Phase Approach

Phase I:
Identify Candidate Computation Decompositions

This approach leads to a small number of implementations, in the tens of variants.

Phase II:
For a given computation decomposition, identify candidates for different memory hierarchy levels

Phase III/IV: Code Gen & Autotuning
Holding computation and data partition sizes fixed (tile parameters), determine most profitable data placement for each data structure. Then, tune for tile parameters.

THE UNIVERSITY OF UTAH

3b. TSG: Identifying Strategies

Parallel Mapping

- Use dependence information and global memory coalescing concept to identify candidate parallel loops for blocks and threads.
- Generate tiling for block and thread decomposition.

Manage Heterogeneous Memory Hierarchy

- Register candidate: any data with reuse inside a thread.
- Shared memory candidate: any data with reuse across threads in a block AND any data with non-coalesced global memory accesses.
- Texture memory candidate: any read-only data already mapped to shared memory or registers.
- Constant memory candidate: read-only data with reuse across threads (and blocks) and short reuse distances.

Other Optimizations

- Aggressive loop unrolling by default to improve ILP, increase register reuse and reduce loop overhead.

3c. Matrix Multiply Example, Portability to Different Architectures

- Next we show the distinct implementations our autotuning framework identifies for the GTX-280 and Fermi
- Achieves comparable results to best-known previous implementations

3c. Comparison of Two GPUs

	GTX-280	Tesla C2050
#SMs	30	14
Cores/SM	8	32
Total cores	240	448
Peak (SP)	933 GF/s	1.03 TF/s
Peak (DP)	87 GF/s	515 GF/s
Global memory	1 GB	3 GB
Bandwidth	142 GB/s	144 GB/s
Shared memory/SM	16KB	(up to) 48 KB
Registers/SM	16 K	32 K
"Texture" accesses	Yes	Yes
Data cache	0	(up to) 48 KB

3c. Automatically-Generated Matrix-Matrix Multiply Scripts

GTX-280 implementation
Mostly corresponds to CUBLAS 2.x and Volkov's SC08 paper

```
1 tile_by_index({"i","j"},{TI,TJ},{l1_control="ii",l2_control="jj"},
                {"ii","jj","i","j"})
2 tile_by_index({"k"},{TK},{l1_control="kk"},
                {"ii","jj","kk","i","j","k"},strided)
3 tile_by_index({"i"},{TJ},{l1_control="tt",l1_tile="t"},
                {"ii","jj","kk","t","tt","j","k"})
4 cudaize("mm_GPU",{a=N*N,b=N*N,c=N*N},
          {block={"ii","jj"},thread={"t","tt"}})
5 copy_to_shared("tx","b",-16)
6 copy_to_registers("kk","c")
7 copy_to_texture("b")
8 unroll_to_depth(2)
```

TC2050 Fermi implementation
Mostly corresponds to CUBLAS 3.2 and MAGMA

```
1  tile_by_index({"i","j"},{TI,TJ},{l1_control="ii",l2_control="jj"},
                 {"ii","jj","i","j"})
2  tile_by_index({"k"},{TK},{l1_control="kk"},
                 {"ii","jj","kk","i","j","k"},strided)
3  tile_by_index({"i"},{TK},{l1_control="t",l1_tile="tt"},
                 {"ii","jj","kk","tt","t","j","k"})
4  tile_by_index({"j"},{TK},{l1_control="s",l1_tile="ss"},
                 {"ii","jj","kk","tt","t","ss","s","k"})
5  cudaize("mm_GPU",{a=N*N,b=N*N,c=N*N},
           {block={"ii","jj"},thread={"tt","ss"}})
6  copy_to_shared("tx","b",-16)
7  copy_to_texture("b")
8  copy_to_shared("tx","a",-16)
9  copy_to_texture("a")
10 copy_to_registers("kk","c")
11 unroll_to_depth(2)
```

Different computation decomposition leads to additional tile command

a in shared memory, both a and b are read through texture memory

THE UNIVERSITY OF UTAH

3d. Another Example, MRI-Q

Source code from Parboil benchmark

```
struct kValues {
 float Kx;
 float Ky;
 float Kz;
 float PhiMag;
} kVals[M];

float x[N], y[N], z[N], Qr[N], Qi[N];
for ( i = 0; i < M; i++) {
  for ( j = 0; j < N; j++) {
    expArg = PIx2 * (kVals[i].Kx*x[j] +
             kVals[i].Ky*y[j] + kVals[i].Kz*z[j]);
    cosArg = cosf(expArg);
    sinArg = sinf(expArg);
    phi = kVals[i].PhiMag;
    Qr[j] += phi * cosArg;
    Qi[j] += phi * sinArg;
  }
}
```

- Kernel from an MRI computation, part of Parboil suite
- Our compiler identified a different strategy that leads to higher performance than the manually-written code

Automatically-generated script

```
1  permute(0,{"j","i"})
2  tile_by_index({"j"},{TI},{l1_control="jj"},{"jj","j","i"})
3  normalize_index("j")
4  normalize_index("i")
5  tile_by_index({"i"},{TJ},{l1_control="ii",l1_tile="i"},
                 {"jj","ii","j","i"})
6  cudaize("kernel_GPU",{x=N,y=N,z=N,Qr=N,Qi=N,kVals=M},
          {block={"jj"},thread={"j"}})
5  copy_to_shared("tx","kVals",1)
6  copy_to_texture("kVals)
7  copy_to_registers("ii","x")
8  copy_to_registers("ii","y")
9  copy_to_registers("ii","z")
10 copy_to_registers("ii","Qi")
11 copy_to_registers("ii","Qr")
```


THE UNIVERSITY OF UTAH

3d. Another Example, 2D Convolution

Sequential Code

```
for(i=0;i<N;++i)
   for(j=0;j<N;++j)
     for(k=0;k<M;++k)
        for(l=0;l<M;++l)
             c[i][j] = c[i][j] + a[k+i][l+j] * b[k][l];
```

Complex bounds for shared memory copy loops

CUDA-CHiLL Recipe

```
N=4096, M=32, TI =32, TJ = 16, Tl=4
permute(0,{"j","i","k","l"})

tile_by_index({"j","i"}, {TI,TJ}, {l1_control="jj",
      l2_control="ii"}, {"jj", "ii", "j", "i","k","l"})

normalize_index("j")
normalize_index("i")
cudaize("Kernel_GPU", {a=(N+M
      c=(N+M)*(N+M)},{block={"jj
      {"j","i"}})
copy_to_shared("tx","a",-16)
copy_to_shared("tx","b",-16)
copy_to_registers("tx", "c")
unroll_to_depth(1)
```

Data structures for shared memory

Optimized Code

```
__shared__ float (_P1[47])[31];
 __shared__ float (_P2[16])[17];float tmp3;

for (tmp = 16 * by + 3 * ty; tmp <= min(16 * by + 30, 16
      * by + 3 * ty + 2); tmp++)
      for (tx1 = 2 * tx; tx1 <= min(2 * tx + 1, 46); tx1++)
           _P1[ tx1][tmp - 16 * by] = a[tmp][32 * bx + tx1];
__syncthreads();
   for (tmp = 0; tmp <= 15; tmp++)
      for (tx1 = 2 * tx; tx1 <= 2 * tx + 1; tx1++)
           _P2[tx1][tmp] = b[tmp][tx1];
 syncthreads();
 mp3 = c[k + 16 * by][tx + 32 * bx];
for (k = 0; k <= 15; k++)
      for (l = 0; l <= 15; l++)
         tmp3 = tmp3 + _P1[l + tx ][k + ty] * _P2[l][k];
c[k + 16 * by][tx + 32 * bx] = tmp3;
```

4. Increased Productivity of CUDA-CHiLL Interface

Simple example where tile size evenly divides problem size.

CUDA-CHiLL script (in Lua):

```
N = 1024
TI=32
tile_by_index({"i"}, {TI},
    {l1_control="ii"},{"ii", "i", "j"})
//normalize to start at "i"=0
normalize_index("i")
cudaize("mv_GPU", {a=N, b=N,
    c=N*N},{block={"ii"}, thread=
    {"i"}})
copy_to_shared("tx", "b", 1)
copy_to_registers("k", "a")
unroll_to_depth(1)
```

Equivalent CHiLL script (generated by Lua interface):

```
NI = 1024
TI = 32
original()
tile(0, 1, TI, 1, i, ii, 1)
tile(0, 3, TI, 2, j, k, 1)
tile(0, 3, 3)
datacopy(0, 3, b,{"tmp1","tmp2"},
    false,0,1,1,true)
add_sync(0, "i")
add_sync(1, "i")
tile(1, 3, 3)
datacopy_privatized(0, k, a, {"i", "j"})
unroll(0, 5, 0)
```

4. Increased Productivity: Matrix Multiply

Low Level

```
permute(0, {i, j})
tile(0, 1, 64, 1, i, ii, 1)
permute(0, {ii, i, j})
tile(0, 3, 16, 2, j, jj, 1)
permute(0, {ii, jj, i, j})
permute(0, {ii, jj, i, j, k})
tile(0, 5, 16, 3, k, kk, 0)
permute(0, {ii, jj, kk, i, j, k})
permute(0, {ii, jj, kk, i, j, k})
tile(0, 4, 16, 4, t, tt, 1)
tile(0, 5, 5)
tile(0, 5, 4)
permute(0, {ii, jj, kk, t, tt, j, k})
cudaize("mm_GPU", {a=N*N, b=N*N, c=N*N},
        {block={"ii","jj"}, thread={"t","tt"}})
datacopy_privatized(0, kk, c, {tx, ty})
unroll(1, 5, 0)
unroll(2, 5, 0)
datacopy(0, 4, b, {tmp1, tmp2}, false,
         0, 1, -16, shared_mem)
add_sync(0, tx)
normalize(7, 5
tile(7, 4, 4, 4, tmp, ty, counted)
tile(7, 6, 14, tx, tx, counted)
add_sync(7, tx)
unroll(0,8,0)
unroll(3,5,0)
unroll(4,5,0)
unroll(5,5,0)
unroll(6,5,0)
unroll(7,6,0)
unroll(0,9,0)
unroll(8,8,0)
unroll(8,9,0)
unroll(30,9,0)
unroll(46,9,0)
unroll(59,9,0)
unroll(72,9,0)
```

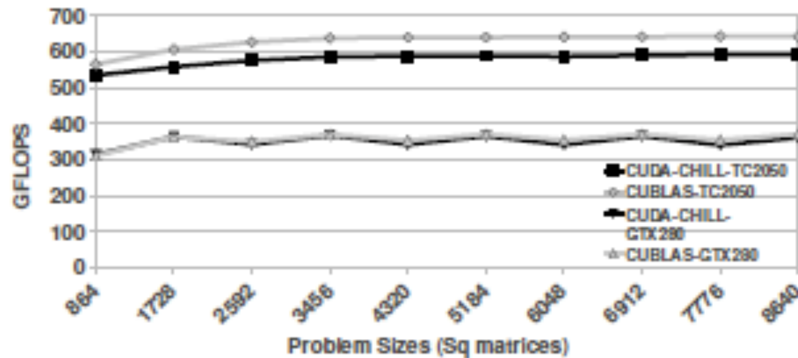
High Level with Abstraction Layer

```
tile_by_index({"i","j"}, {TI,TJ},
              {l1_control="ii", l2_control="jj"},
              {"ii", "jj", "i", "j"})

tile_by_index({"k"}, {TK},
              {l1_control="kk"},
              {"ii", "jj", "kk","i", "j","k"},
              strided)

tile_by_index({"i"}, {TJ},
              {l1_control="tt",l1_tile="t"},
              {"ii", "jj", "kk","t","tt","j","k"})

cudaize("mm_GPU", {a=N*N, b=N*N, c=N*N},
        {block={"ii","jj"}, thread={"t","tt"}})

copy_to_registers("kk", "c")
copy_to_shared("tx", "b", -16)

unroll_to_depth(2)
```

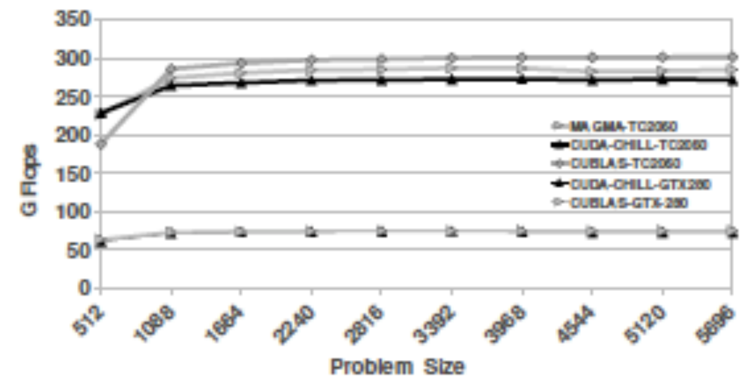
CHiLL Script Source: Gabe Rudy

THE UNIVERSITY OF UTAH

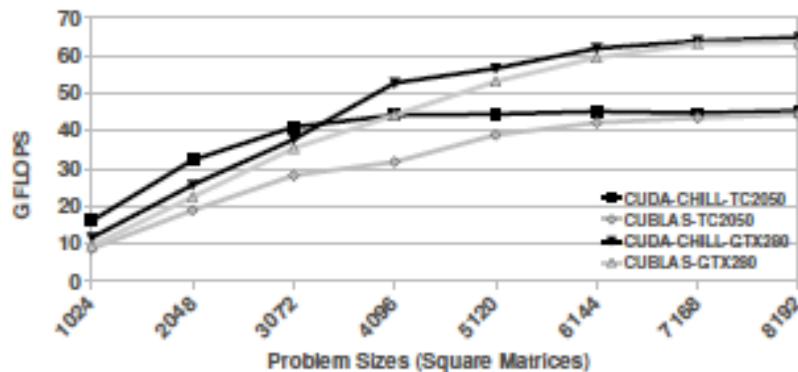
5. Performance Results: Matrix-Matrix and Matrix-Vector Multiply



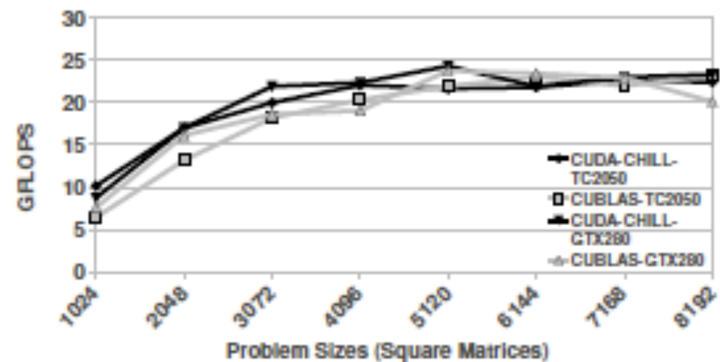
(a)SGEMM GFlops



(b)DGEMM GFlops



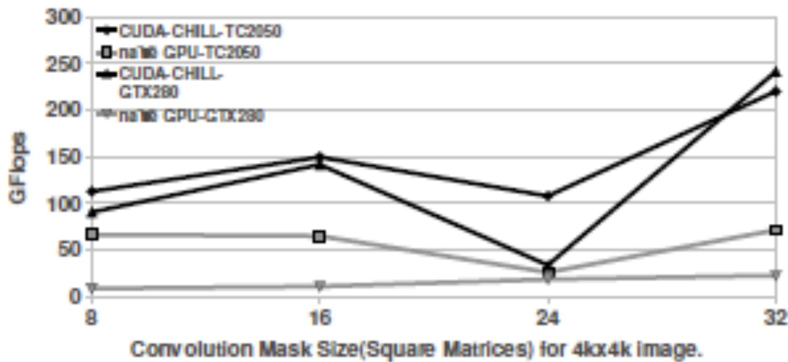
(c)SGEMV GFlops



(d)DGEMV GFlops

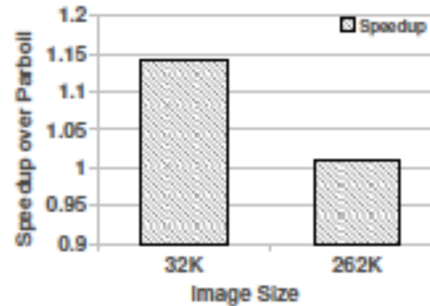
THE UNIVERSITY OF UTAH

5. Performance Results: Convolution and MRI-Q

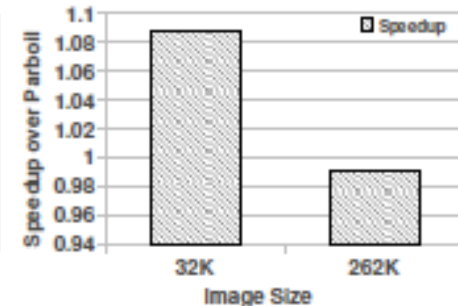


2-D Convolution

MRI-Q



TC2050 Fermi



GTX-280

THE UNIVERSITY OF UTAH

5. Performance Results: Impact of Different Memory Hierarchy Levels

	Unroll		Registers		Shared Memory		Texture Memory	
Kernel	GTX280	Tesla C2050	GTX280	Tesla C2050	GTX280	Tesla C2050	GTX280	Tesla C2050
DGEMM	94.85	96.55	92.32	84.86	72.17	58.71	1.38	4.77
DGEMV	57.55	5.55	63.63	68.99	27.16	0.38	0.0	3.8
DGEMV(T)	64.31	0.63	20.51	19.94	88.83	63.06	0.55	0.0
MRIQ	0.00	1.63	12.47	11.25	10.61	12.89	0.0	0.0
Conv.	44.16	47.46	77.29	54.29	89.19	23.24	5.66	0.0

- Table shows reduction in performance when specific level of the memory hierarchy is not used.

5. Performance Results: A Few More Details

- Generated code is very long, over 1000 lines of code for some versions of Matrix-Matrix Multiply
  - Problem size may not be evenly divided by computation decomposition parameters
  - Cleanup code for tiling and unrolling can be lengthy, important to optimize too
- No more than 32 versions needed to generalize

Summary of Lecture

- Two ideas
  - A programming language interface to CHiLL's primitives allows custom higher-level abstractions
  - Used to develop CUDA-CHiLL abstractions for auto-tuning high-performance GPU code
- Features
  - TSG generates multiple possible implementations that are searched using autotuning
  - Performance: can sometimes outperform CUBLAS and manual code

References

Other GPU compiler frameworks.

M. Baskaran, J. Ramanujam, and P. Sadayappan, "Automatic C-to-CUDA Code Generation for Affine Programs ," International Conference on Compiler Construction (CC), March 2010.

H. Cui, L. Wang, J. Xue, Y. Yang, X. Feng, Automatic Library Generation for BLAS3 on GPUs, Proceedings of the IEEE International Parallel Distributed Processing Symposium, May 2011.

T. D. Han and T. S. Abdelrahman, "hiCUDA: High-Level GPGPU Programming," IEEE Transactions on Parallel and Distributed Systems, vol. 22, no. 1, pp. 78-90, Jan. 2011.

S. Lee and R. Eigenmann, "OpenMPC: Extended OpenMP Programming and Tuning for GPUs". Proceedings of the 2010 ACM/IEEE Conference on Supercomputing, Nov. 2010.

Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU Compiler for Memory Optimization and Parallelism Management", Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI2010), June, 2010.

CUDA-CHiLL reference.

G. Rudy, M. Khan, M. Hall, C. Chen, and J. Chame. 2010. A programming language interface to describe transformations and code generation. In Proceedings of the 23rd international conference on Languages and compilers for parallel computing (LCPC'10), Keith Cooper, John Mellor-Crummey, and Vivek Sarkar (Eds.). Springer-Verlag Publishers, 136-150.