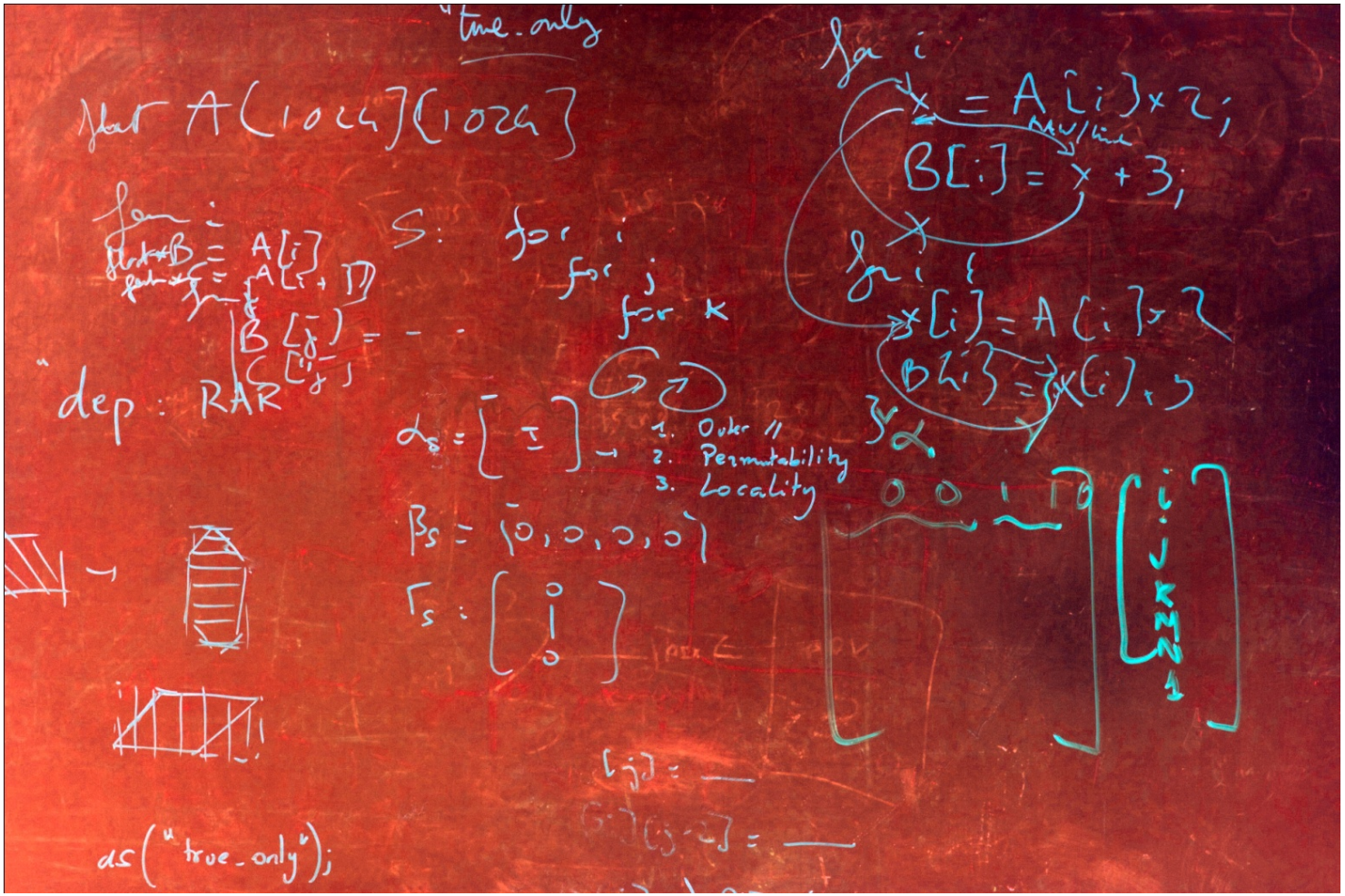
A Programming Model and Tool for Automatic High-Performance C to CUDA Mapping

B. Meister, A. Leung, M. Baskaran, N. Vasilache, A. Hartono, A. Johnson, R. Lethin



The (general) problem

Increasingly complex application design

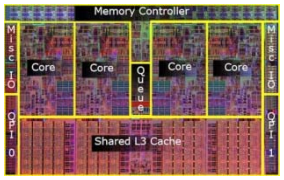
- Bigger problem sizes
- Higher dimensionality
- More sub-problems
- Proprietary programming models and languages (e.g., CUDA)

Increasingly complex hardware

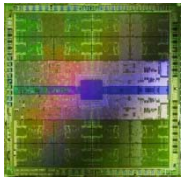
- More coarse-grained parallelism
- SIMD
- Explicit resource management (memories, communication)
- Mixed execution models

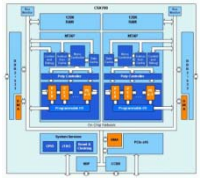
Software challenges

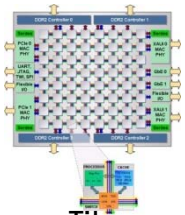
- Productivity
- Performance
- Portability



Intel Nehalem


Nvidia GPU


ClearSpeed


Tilera

Outline

New chips: A major programming issue

Approach: R-Stream, a source-to-source compiler

Mapping to CUDA

- Highlights
- GPU

Performance results

- GPU
- Other targets

Approach: R-Stream, an Automatic C Parallelizer

Source-to-source auto-parallelizing compiler

Takes in sequential code, combinations of loops in C

Addresses specific features of emerging architectures

- Lots of parallelism, granularities, hierarchical mixed execution models, explicit management of memories, explicit bulk communications, asynchronous comm., importance of locality

  Reduces programming effort

- User writes C code in a clean, "textbook" style
- Functions to be processed marked with `#pragma rstream map`

  Produces *mapped* C code to be compiled by a backend compiler (GCC, ICC, nvcc, etc.)

Does this work ?

We currently map programs to a variety of targets

- Tilera, SMP, nVidia GPU, Clearspeed, Cell, SGI RC100

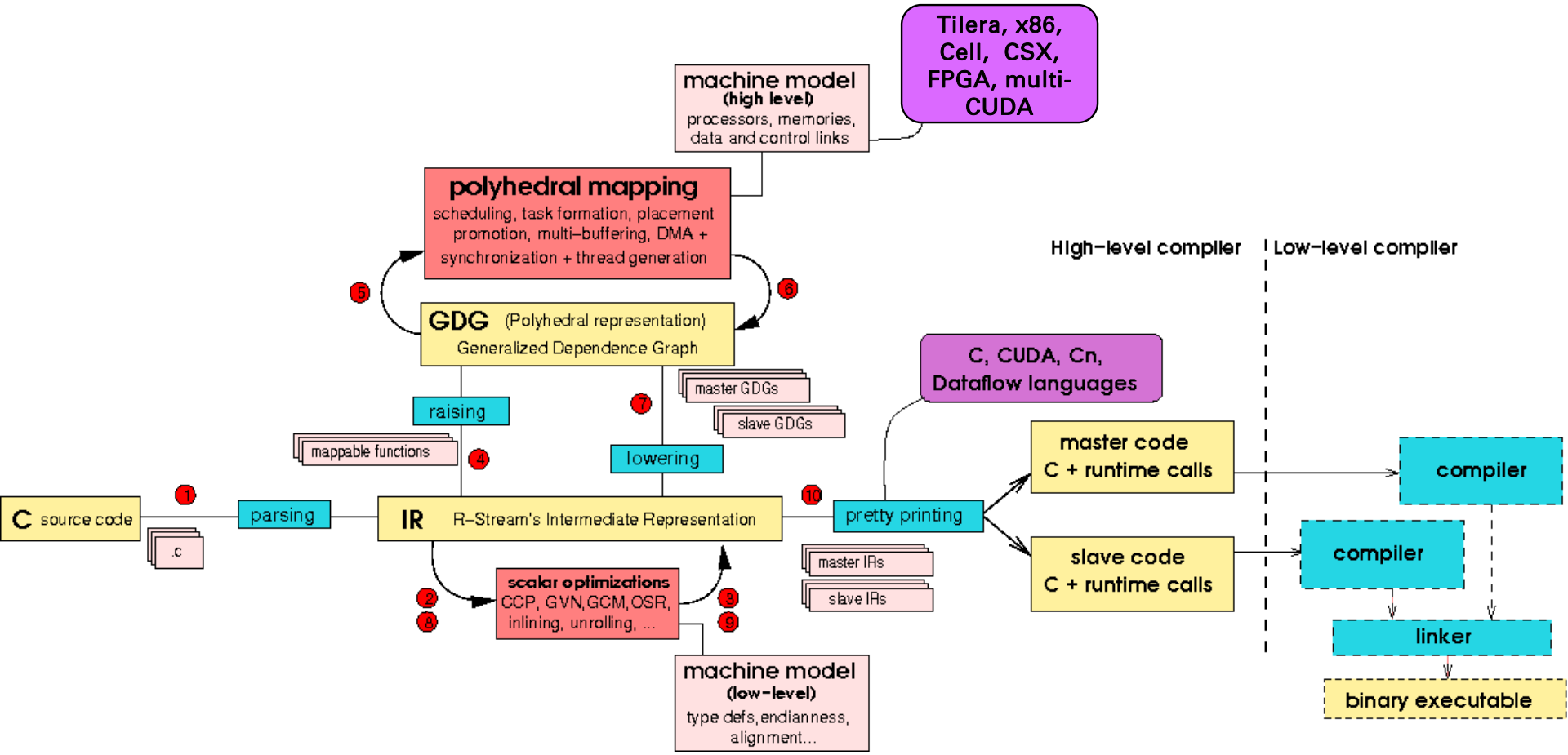
Many times, performance in the ball park of libraries, sometimes better

- Depends on maturity of target-specific optimizations
- More sophisticated mappings than your regular programmers'
- Pathological cases exist
- If library call is better: R-Stream handles library calls

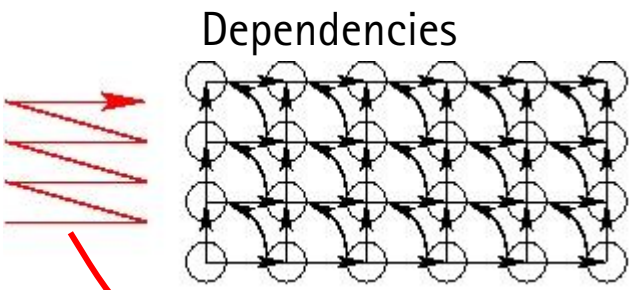
Application domain large for those mapping capabilities

- Supports dynamic programs, library calls, etc.
- But the more static information, the more precise the mapping

R-Stream: overview of the compilation process

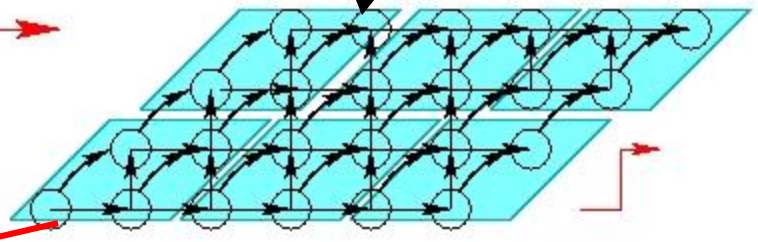


(Very) rough overview of the mapping process

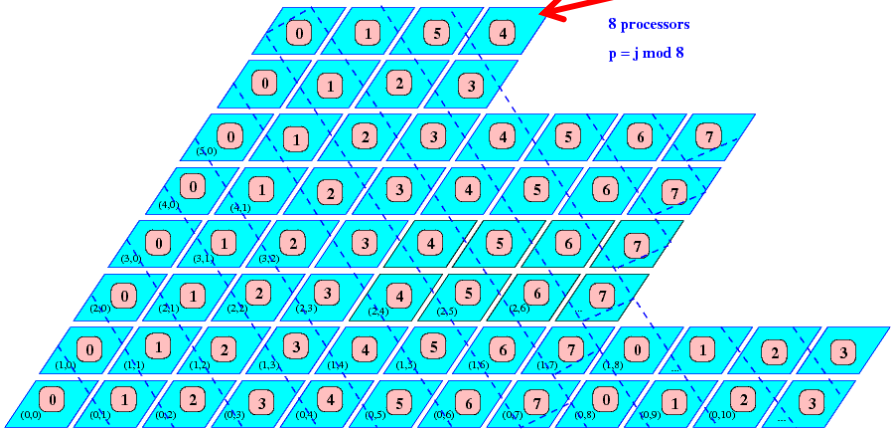


1- Scheduling:
Parallelism, locality, tilability

2- Task formation:
- Coarse-grain atomic tasks
- Master/slave side operations



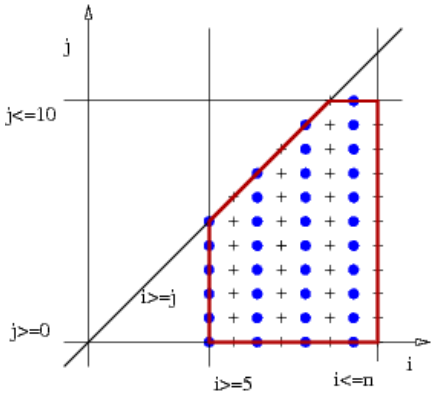
3- Placement:
Assign tasks to blocks/threads

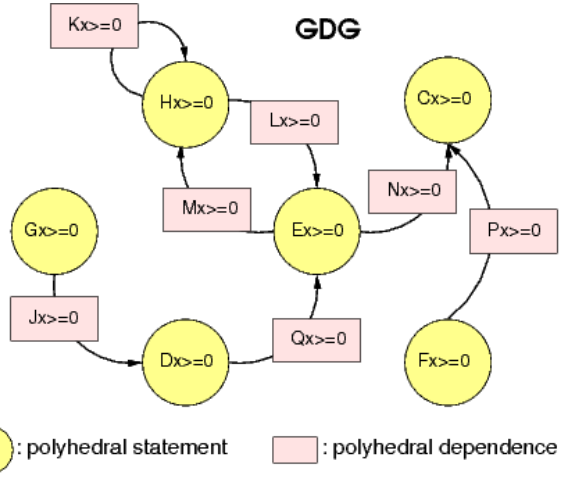


- Local / global data layout optimization
- Multi-buffering (explicitly managed)
- Synchronization (barriers)
- Bulk communications
- Thread generation -> master/slave
- CUDA-specific optimizations

R-Stream: Style-based programming and polyhedral IR for loop-based code

```
n = f();
for (i=5; i<= n; i+=2) {
  A[i][i] = A[i][i]/B[i];
  for (j=0; j<=i; j++) {
    if (j<=10) {
      … A[i+2j+n][i+3]…
  }
}
```





$$\{i, j \in Z^2 \mid \exists k \in Z, 5 \le i \le n; 0 \le j \le i; j \le i; i = 2k+1\}$$

$$\begin{pmatrix} A_0 \\ A_1 \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 2 & 1 & 0 \\ 1 & 0 & 0 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ n \\ 1 \end{bmatrix}$$

Affine and non-affine transformations
Order and place of operations and data

Loop code represented (exactly *or conservatively*) with polyhedrons
→ High-level, mathematical view of a mapping
→ But targets concrete properties: parallelism, locality, memory footprint

Outline

New chips: A major programming issue

Approach: R-Stream, a source-to-source compiler

Mapping to CUDA

- Highlights
- GPU

General performance results

CUDA - What is needed

Expressing the program in CUDA

Follow accelerator execution model

- Define master and slave computations and partition codes
- Generate (CUDA) kernel configuration, communications, kernel launches
- Parallelize across multiple GPUs

Doing all this by hand is costly, error-prone, performance is not portable

Promise of our technology

- Meet or exceed what you can do by hand
- Without knowing much about CUDA or GPUs (or other multicore targets)

Main goals of the mapping process

Those you find in any application porting paper

- Coalescing of transfers from/to global memory
- Data footprint has to fit in shared, global, private memories
- High occupancy of threads and blocks
- Minimize data transfers
- Avoid shared memory bank conflicts

Distinctive parts of mapping process

Tradeoff among parallelism, locality and coalesced accesses

Advanced task formation and placement

Coalesced data movement

Hierarchical mapping

Trading off parallelism, locality and coalescing

Affine scheduling algorithm:

- Fuses loops to increase computational intensity and locality
- while exposing enough parallel iterations for threads and blocks for occupancy
- and exposing parallel iterations that enable coalescing

Adaptive array expansion algorithm duplicates data just enough to expose the desired parallelism

Tradeoff Example

Array z gets expanded, to introduce another level of parallelism

Maximum fission destroys locality

```
/*
 * Original code:
 */
for (k=0; k<400; k++) {
 for (i=0; i<3997; i++) {
  z[i]=0;
  for (j=0; j<4000; j++)
   z[i]= z[i]+B[i][j]*x[k][j];
 }
 for (i=0; i<3997; i++)
  w[i]=w[i]+z[i];
}
```

Coalescing along i

Max. parallelism (no fusion)

```
doall (i=0; i<400; i++)
 doall (j=0; j<3997; j++)
  z_e[j][i]=0
doall (i=0; i<400; i++)
 doall (j=0; j<3997; j++)
  for (k=0; k<4000; k++)
   z_e[j][i]=z_e[j][i]+B[j][k]*x[i][k];
doall (i=0; i<3997; i++)
 for (j=0; j<400; j++)
  w[i]=w[i]+z_e[i][j];
doall (i=0; i<3997; i++)
 z[i] = z_e[i][399];
```

Data accumulation

→ 2 levels of parallelism, but poor data reuse (on array z_e)

Tradeoff Example (cont.)

```
/*
  * Original code:
  */
for (k=0; k<400; k++) {
  for (i=0; i<3997; i++) {
    z[i]=0;
    for (j=0; j<4000; j++)
      z[i]= z[i]+B[i][j]*x[k][j];
  }
  for (i=0; i<3997; i++)
    w[i]=w[i]+z[i];
}
```

Max. fusion and coalescing

Coalescing along i

```
doall (i=0; i<3997; i++)
  for (j=0; j<400; j++) {
    z[i]=0;
    for (k=0; k<4000; k++)
      z[i]=z[i]+B[i][k]*x[j][k];
    w[i]=w[i]+z[i];
  }
```

Aggressive loop fusion destroys parallelism (i.e., only 1 degree of parallelism)

→ Very good data reuse (on array z), but only 1 level of parallelism

Tradeoff Example (cont.)

```
/*
  * Original code:
  */
for (k=0; k<400; k++) {
  for (i=0; i<3997; i++) {
    z[i]=0;
    for (j=0; j<4000; j++)
      z[i]= z[i]+B[i][j]*x[k][j];
  }
  for (i=0; i<3997; i++)
    w[i]=w[i]+z[i];
}
```

Parallelism with partial fusion and coalescing →

Expansion of array z

Coalescing along j

Partial fusion doesn't decrease parallelism

```
doall (i=0; i<3997; i++) {
  doall (j=0; j<400; j++) {
    z_e[i][j]=0;
    for (k=0; k<4000; k++)
      z_e[i][j]=z_e[i][j]+B[i][k]*x[j][k];
  }
  for (j=0; j<400; j++)
    w[i]=w[i]+z_e[i][j];
}
doall (i=0; i<3997; i++)
  z[i]=z_e[i][399];
```

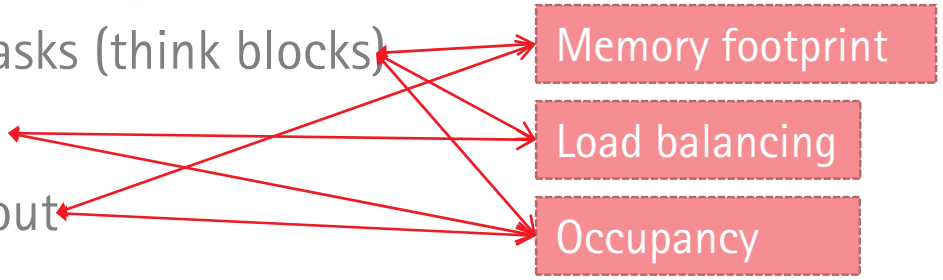
Data accumulation

→ 2 levels of parallelism with good data reuse (on array z_e)

Advanced task formation and placement

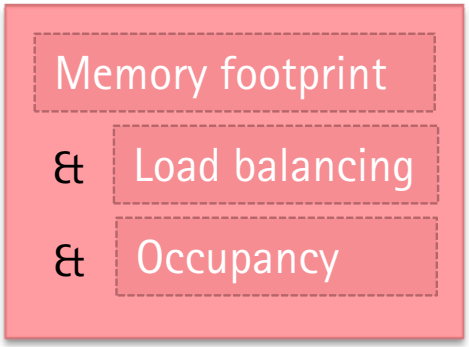
Natural phase ordering problem among:

- Forming atomic tasks (think blocks)
- Distributing tasks
- Defining data layout



R-Stream components interact with each other

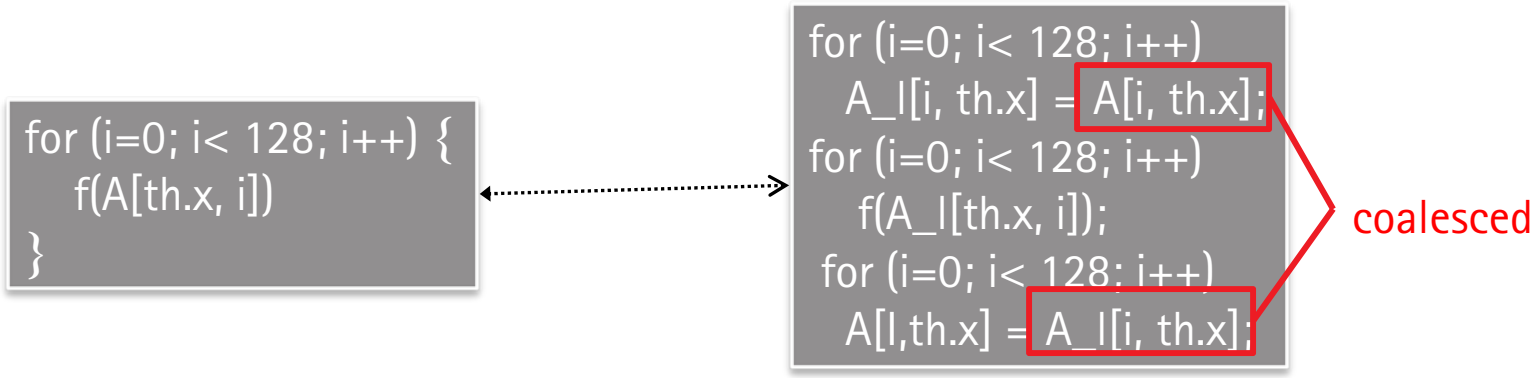
- Forward and Backward (feedback)
- Produces "sensible" mappings:
  - As many constraints as possible get satisfied at once



Coalesced data movement

Data that are not coalesced naturally

- Can be loaded to shared memory in a coalesced way

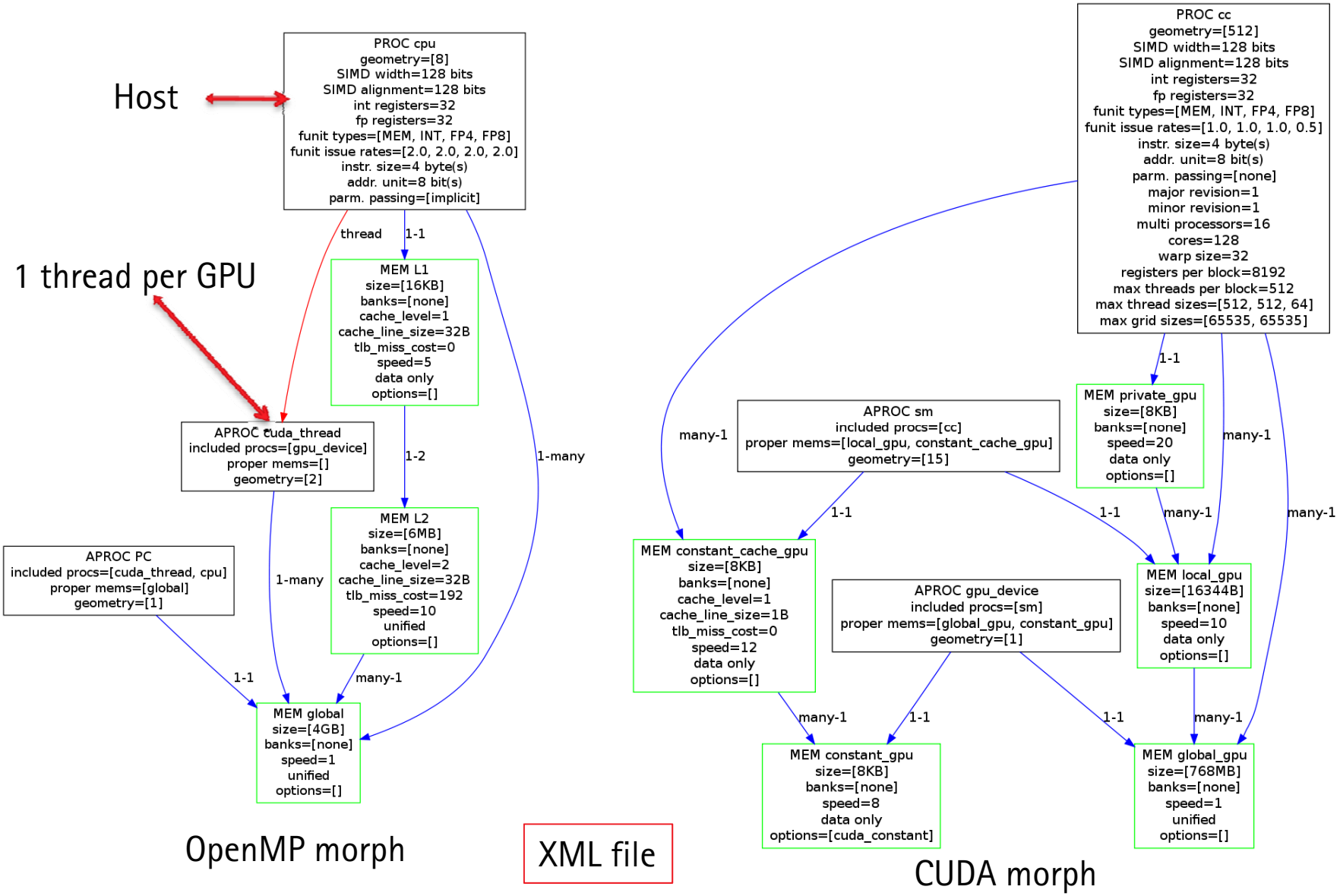


Multi-GPU - Hierarchical mapping

R-Stream mapping is driven by a machine model

- Describes targeted machine as a graph of processors, memories, explicit data links, etc.
- Hierarchical mappings: decompose the problem using "morphs", i.e., views of the machine
- A backend is associated with each morph
  - Defines the way code is generated (OpenMP, CUDA, etc.)

Machine model example: multi-Tesla



Performance Results

Benchmarks

- Stencil kernels with multiple time iterations
  - Gauss-seidel (2D – 5 points and 9 points stencil)
- Stencil kernels with single time iteration
  - Divergence (3D)
  - Gradient (3D)
  - Laplacian (3D)
  - RTM (3D)

Performance Results

Stencil kernels with single time iteration

- Double Precision Performance
- Problem size: 256^3

Kernel	Performance (Gflops)	
	GTX 285	GTX 480
Divergence	15.59	28.74
Gradient	8.02	17.55
Laplacian	16.79	41.33
RTM	24.69	50.74

Performance Results

Stencil kernels with multiple time iterations

- Single Precision (SP) and Double Precision (DP) Performance
- Problem size: 4096^2

Kernel	Performance (Gflops)			
	GTX 285		GTX 480	
	SP	DP	SP	DP
Gauss-seidel_5pt	16.41	8.34	17.04	11.98
Gauss-seidel_9pt	21.51	8.40	24.87	18.40

Outline

New chips: A major programming issue

Approach: R-Stream, a source-to-source compiler

Mapping to CUDA

- Highlights
- GPU

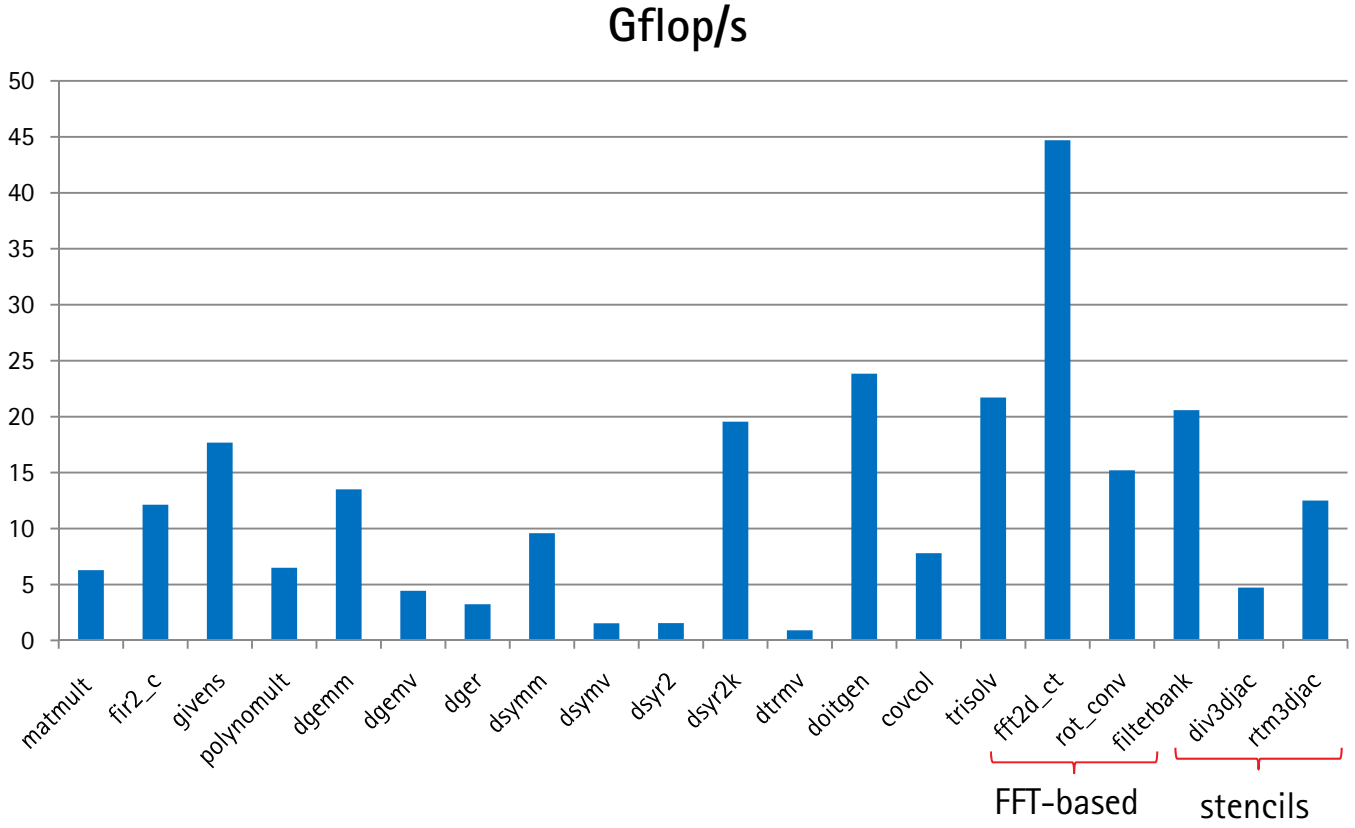
General performance results

Conclusion

x86 multicore

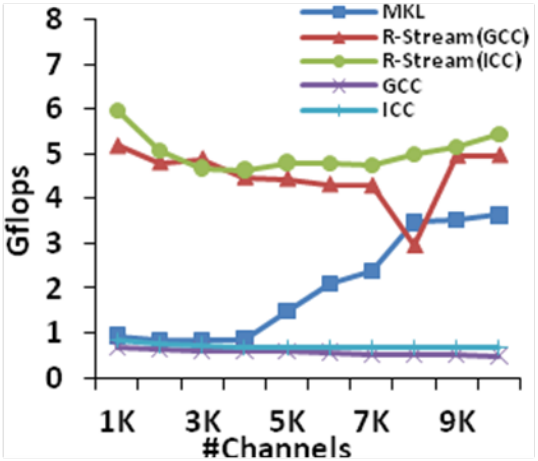
Dual E5405 Xeon 2.0 GHz with 9 GB. R-Stream 3.1.2

GCC 4.3.0 ("-O6 -fno-trapping-math -ftree-vectorize -msse3 -fopenmp")

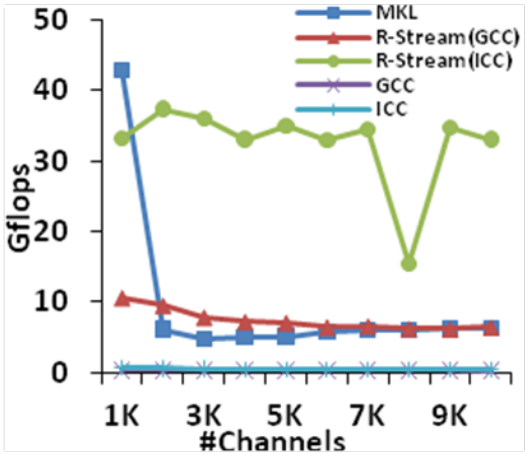


x86 multicore (cont.)

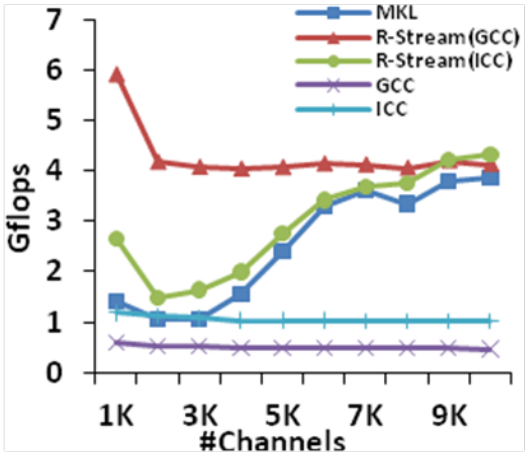
Comparison against Intel's MKL on a few radar applications:



MVDR-SER



CSLC-LMS



CSLC-RLS

Dual E5405 Xeon 2.0 GHz with 9 GB, Linux 2.6.25 (x86-64). R-Stream 3.1.2
GCC 4.3.0 ("-O6 -fno-trapping-math -ftree-vectorize -msse3 -fopenmp" flags)
ICC 11.0 (with "-fast -openmp" flags).
Intel MKL 10.2.1.

Tilera results

Integer

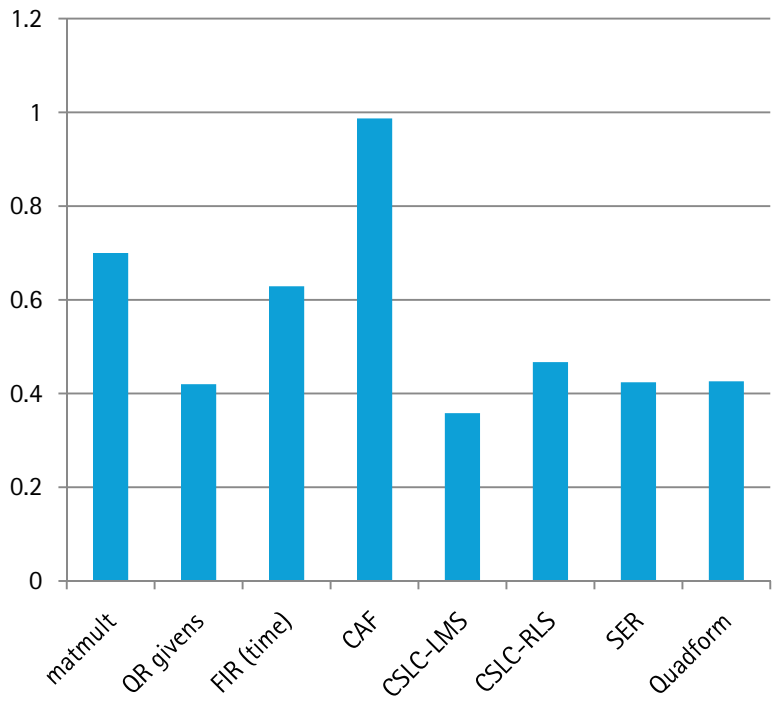
Polynomial multiplication
N=32768, 7x8 tiles

R-Stream	Locality opt	Gop/s	Speedup
N	N	0.048	1
Y	N	2.273	47X
Y	Y	3.416	71.2X

Matrix-matrix multiply
4096x4096, 7x8 tiles

R-Stream	Locality opt	Gop/s	Speedup
N	N	0.013	1
Y	N	0.272	21.3X
Y	Y	8.104	634X

Floating Point (SW Gflop/s)



TileExpressPro-20G

Conclusion

R-Stream simplifies software development and maintenance

Porting: reduces expense and delivery delays

Does this by automatically parallelizing loop code

- While optimizing for data locality, coalescing, etc.

Addresses broad range of applications

- Dense loop-intensive computations

Promise of this technology

- Meet or exceed what you can do by hand

Questions