Multicore Autotuning for Stencil-based PDE Solvers

Cy Chan

NERSC
(in collaboration with
HPCRD: Future Technologies Group)
Lawrence Berkeley National Laboratory

August 21, 2008

Introduction

- High Performance Computing is heading towards a highly parallel future
  - We need to adapt existing scientific programs to properly utilize the new hardware
- Two goals that are seemingly at odds in HPC:
  - Optimize program for many parallel hardware platforms
  - Preserve a maintainable (and human readable) code base
- Possible approaches to autotuning scientific code:
  - Current solution: for each scientific program, write a perl script that generates many tuned versions
  - Our approach: create a framework that will work automatically for many different programs

Autotuning Stencil Kernels

- Our framework is targeted to stencil kernels
- Advantages of working with stencils:
  - Large amount of computational work that is data independent
  - Data access pattern is fixed, so optimizations can be made during array indexing
- Data independence is key:
  - Compilers spend lots of computation doing dependency analysis, affine transformations, etc.
  - We can avoid this step because we know we have a stencil

Our Stencil Autotuner

- Work presented here leverages previous work (Shoaib Kamil) that parses Fortran to produce an abstract syntax tree (AST)
  - Fortran has a clear multidimensional array representation and is commonly used in scientific codes
- Workflow:
  - Parser (Fortran Loop ⇒ AST)
  - AST Transformations (AST ⇒ AST)
  - Code Generation (AST ⇒ Fortan/C/CUDA code)
- Languages used:
  - Parser is written in lex and yacc
    - Outputs Lisp data structure through interface to C supported by Embedded Common Lisp
  - AST Transformations and Code Generation are written in Lisp

Input: Fortran Loop

- Input: Fortran stencil loop:

```
do k = ...
  do j = ...
    do i = ...
      B(i,j,k) = A(i+1, j , k )
               + A( i ,j+1, k )
               + A( i , j ,k+1)
    enddo
  enddo
enddo
```

Output: C Loop (1)

- Output C array indexing (pointer chasing):

```
for (k = ... ) {
  for (j = ... ) {
    for (i = ... ) {
      B[k-1][j-1][i-1] =
           A[k-1][j-1][ i ]
         + A[k-1][ j ][i-1]
         + A[ k ][j-1][i-1];
    }
  }
}
```

Output: C Loop (2)

- Output C preprocessor indexing:

```
#define _A_Index(G1,G2,G3) ((G3)*ny+(G2))*nx+(G1)
#define _B_Index(G1,G2,G3) ((G3)*ny+(G2))*nx+(G1)

for (k = ... ) {
   for (j = ... ) {
      for (i = ... ) {
        B[_B_Index(i-1,j-1,k-1)] =
            A[_A_Index( i ,j-1,k-1)]
          + A[_A_Index(i-1, j ,k-1)]
          + A[_A_Index(i-1,j-1, k )];
      }
   }
}
```

Output: C Loop (3)

- Output C pointer/offset indexing:

```
for (k = ... ) {
   for (j = ... ) {
      double *G1, *G2;
      int G3 = _A_Index(-1,1,0);
      int G4 = _A_Index(-1,0,1);
      for (i = ... ,
           G1 = B + _B_Index(i-1,j-1,k-1),
           G2 = A + _A_Index( i ,j-1,k-1);
           ... ; i++, G1++, G2++) {
         G1[0] = G2[0] + G2[G3] + G2[G4];
      }
   }
}
```

A[_A_Index(i, j-1, k-1)]

A[_A_Index(i-1, j, k-1)]

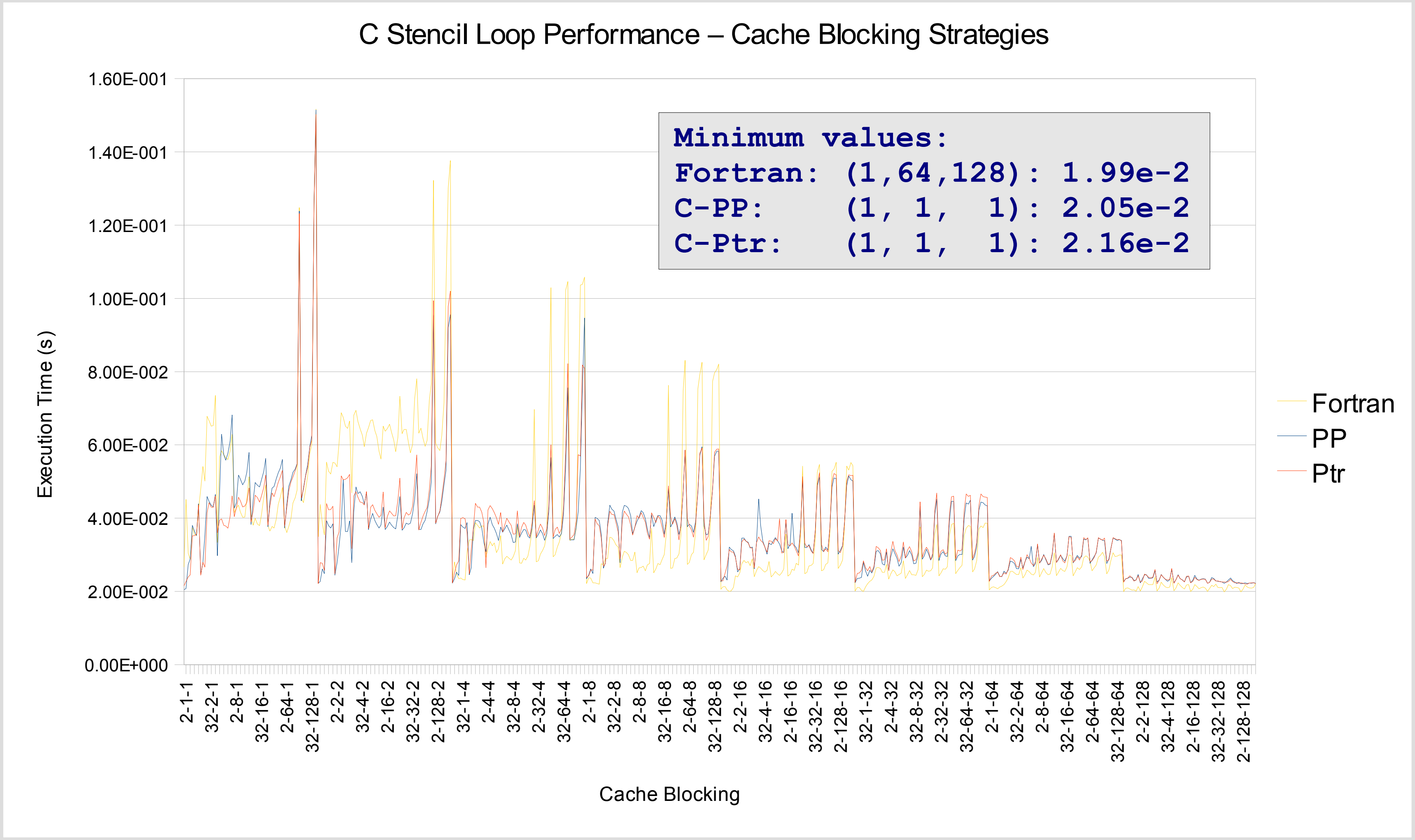
A[_A_Index(i-1, j-1, k)]

C Loop Test Framework

- We can now generate C stencil loops
  - Need to incorporate them into a program
- A template program `#include`s an auto-generated loop
- To test performance, we generated a bunch of loops
  - Applied AST transformations to vary cache blocking strategy
- We ran the template program for each loop to get timing data
- Testbed: NERSC Jacquard
  - 2.2 GHz AMD Opteron

C Loop Performance

7-Point Stencil Probe (N = 128)



```
Minimum values:
Fortran:  (1,64,128): 1.99e-2
C-PP:     (1, 1,  1): 2.05e-2
C-Ptr:    (1, 1,  1): 2.16e-2
```

C Loop Test: Lessons Learned

- Fortran still competitive with optimized C code
  - Best case Fortran code is 8% faster than best case C code
- Performance sensitive to initialization values
  - Initializing values of data block to 1 resulted in early completion of a division in kernel
    - Computed 0 divided by 1
    - Computing 0 divided by X or Y divided by 1 also fast
  - Should initialize data to random numbers!

Stencil Parallelization with CUDA: Domain Decomposition

- Nested for loop specifies an **index block** to be iterated over
- Parallelize by splitting index block into sub-blocks
- Assign sub-blocks to different thread blocks on GPU
- Each thread block uses multiple threads to iterate over its sub-block

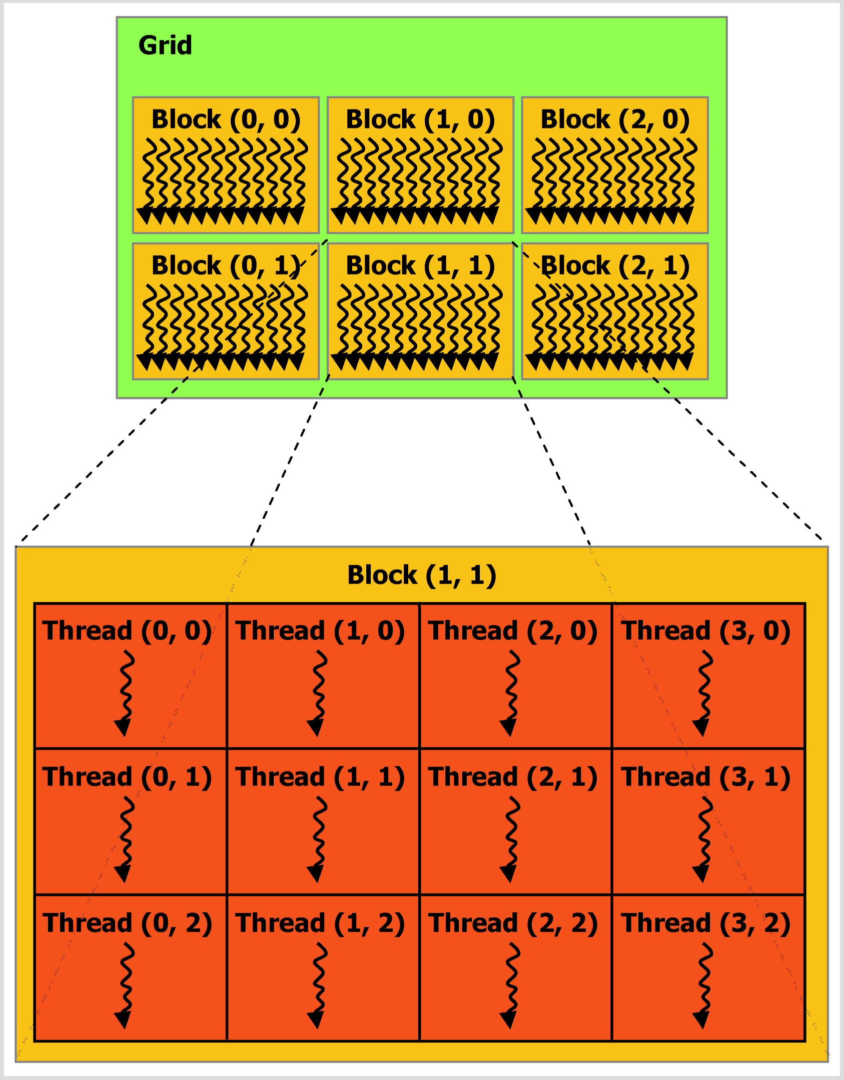


Image: NVIDIA

Stencil Parallelization with CUDA: Memory Hierarchy

- NVIDIA GPUs have a memory hierarchy
- **Global** memory is uncached
- **Shared** memory is on chip but small and inaccessible from host
- Two versions:
  1) global memory only
  2) use shared memory as local store (still under development)

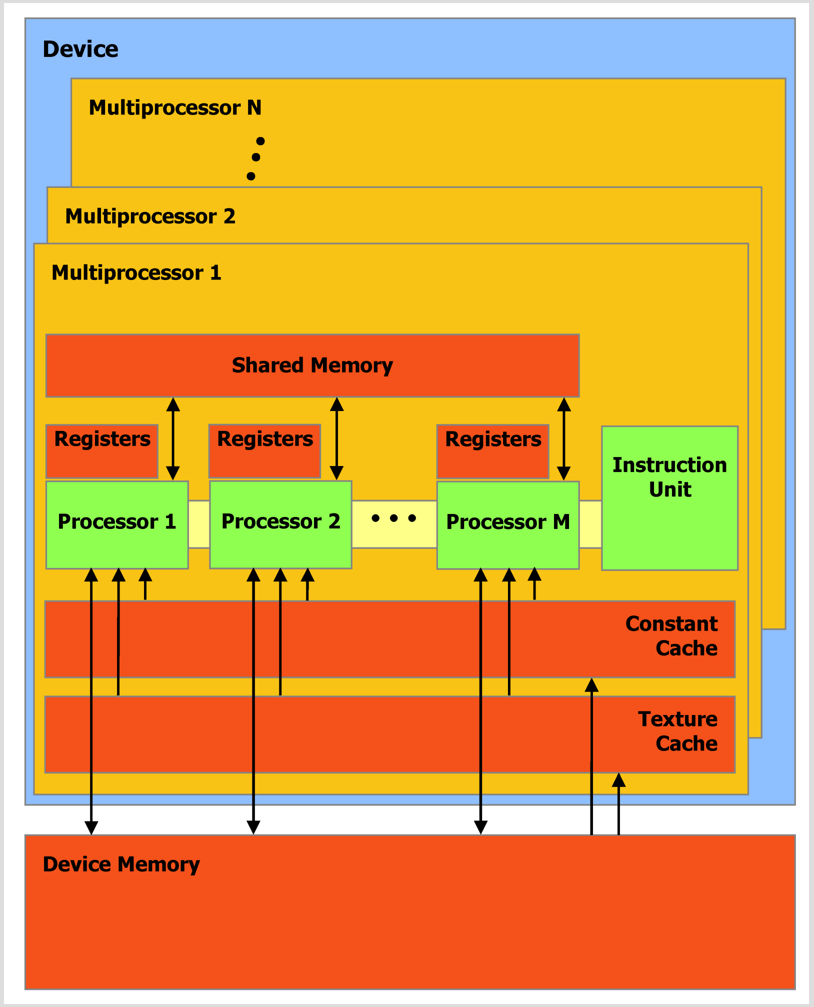


Image: NVIDIA

Requirements for Our CUDA Framework

- Fortran loop must be contained within a time step
  - All calculations in the loop are independent
- Arrays must use loop index variables consistently, e.g.:
  - If `A(i,j)` appears, then `A(j,i)` cannot
- Index expressions are limited to the form:
  - `var + const`
  - necessary to compute extents in each array dimension

CUDA Output

- As with the C code generator, we take a Fortran loop as the input
- Produces two functions:
  - 1) Direct function – runs the stencil kernel on data already residing in device memory
  - 2) Wrapper function – copies an array from host to device, calls the direct function for a user-specified number of iterations, and copies the result back to the host
- Both functions allow the user to specify at runtime:
  - the problem size
  - data and thread blocking strategies

CUDA Code Generation Mechanics

- CUDA language: C with some extensions for GPU
  - Leverage our work with C
- In addition to generating the stencil loop, we need multiple auxiliary code snippets, including:
  - Indexing macros
  - Function definitions and calls
  - Memory management (allocations and copies):
    - host memory ⇔ device global memory ⇔ device shared memory
- Generic functions in template do heavy lifting:
  - Allocations (byte-alignment, dimension data)
  - Copies (coalesced memory access)

Example: Template Wrapper Function

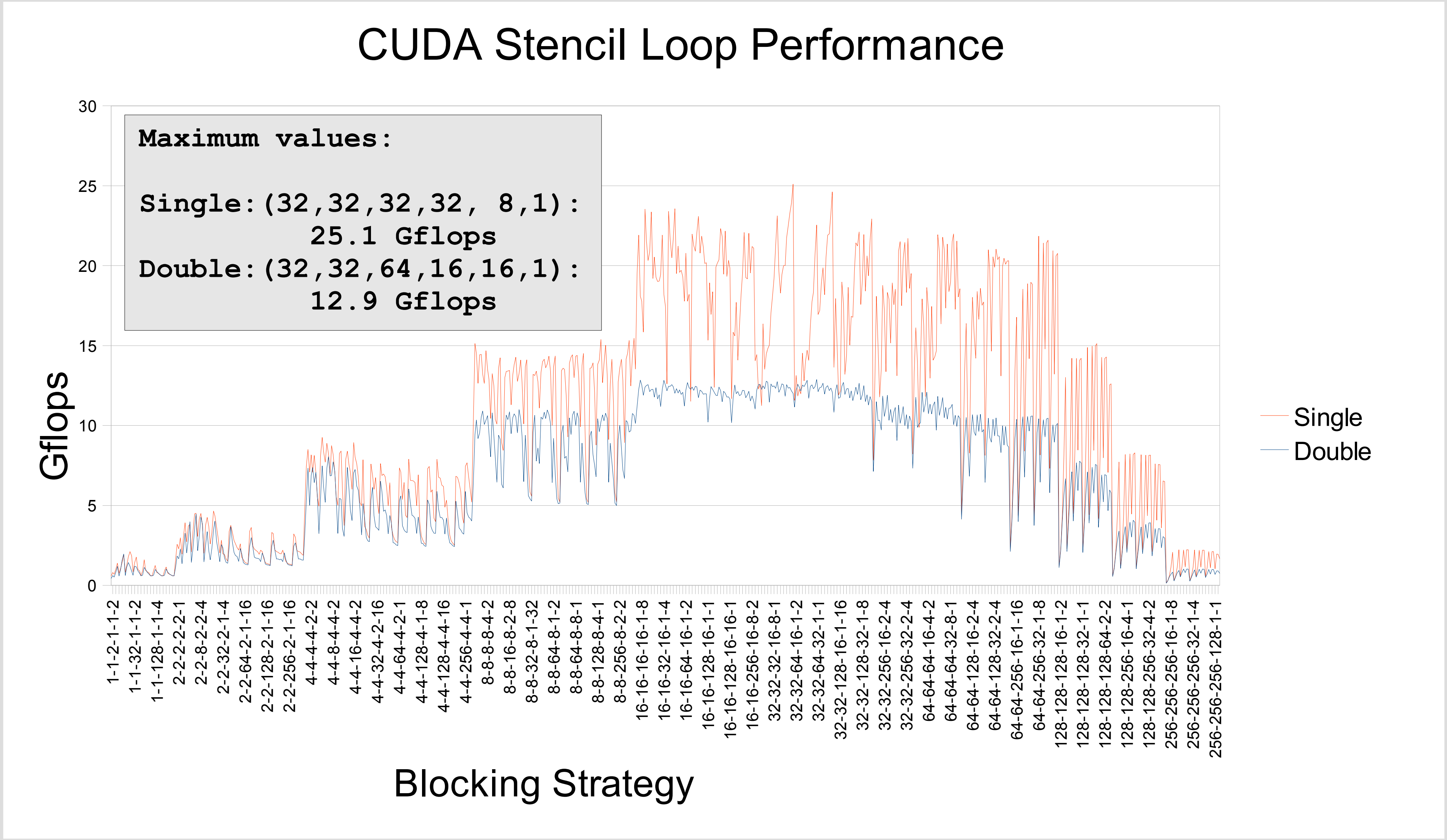
```
void cudaStencilWrapper( ... ) {

#include "kernelSpecific/allocations.c" // allocate device memory for arrays
#include "kernelSpecific/copyIn.c" // copy in arrays to device

    if (copyFlag) {
#include "kernelSpecific/copyToOutput.c" // copy input array to output array
    }

    for (int i = 0; i < numIterations; i++) {
        if (i > 0) {
#include "kernelSpecific/swapArrayPointers.c" // swap input & output ptrs
        }
        cudaStencil(
#include "kernelSpecific/hostFuncCallArgs.c"// call direct stencil function
            indexDim, indexBlockDim, threads);
    }

#include "kernelSpecific/copyOut.c" // copy out arrays from device
#include "kernelSpecific/frees.c" // free device memory

}
```

CUDA Testbed Specifications

- NVIDIA GeForce GTX280 GPU
  - Vector Cores: 30
    - Each core has 8 lane SP MAD and one DP MAD
  - Clock: 1.3 GHz
  - Peak: 624 Gflops SP, 78 Gflops DP
  - Device memory: 1 GB GDDR3
  - Memory bandwidth: 141.7 GB/s, 127 GB/s sustained
  - Host interface (PCIe 2.0): 6GB/s sustained

CUDA Performance

7-Point Heat Equation (N = 256)
NVIDIA GTX280 – Global Memory Only



Maximum values:

Single: (32,32,32,32, 8,1): 25.1 Gflops
Double: (32,32,64,16,16,1): 12.9 Gflops

Comparison Vs. Hand-Tuned 7-Point Heat Equation Code

- Hand-tuned code with hard-coded data and thread block sizes achieves 76 Gflops single precision and 27 Gflops double precision
- Hand-tuned code leverages non-generalizable heuristics for maximum performance
  - Some data are kept in registers instead of shared memory
  - Circular queue is only one plane wide, so no circular addressing required
- Our framework must work for generalized stencils
  - One possible solution: try to recognize special cases and implement applicable heuristics
  - Can default to general case

Future Work

- Finish shared memory version of CUDA code
  - Reduce global device memory communication
  - Increase computational overhead
- Extend code generation framework to support pthreads on SMPs and chip multiprocessors
  - Can use same domain decomposition as CUDA
- Identify cases where we can exploit heuristics used in the hand-tuned version
- Reproduce **all** capabilities of the autotuning strategies previously presented by our group

Summary

- We have built a framework that takes maintainable Fortran code and produces tuned versions of that code in C and CUDA
- This work generalizes existing work (that uses perl scripts) to accommodate a broader range of stencil kernels
- Performance is already reasonably good given the generality of framework