GPGPU Acceleration of MFIX-DEM

John Urbanic

Pittsburgh Supercomputing Center

May 22, 2012

What I hope to communicate…

You've already heard something about MFIX.

I'm going to tell you about our work in progress: accelerating MFIX using GPUs.

In particular, I'm going to focus on our novel approach of using OpenACC to do so.

I believe many of you will find this a compelling technology.

PSC Team:

Nick Nystrom
Anirban Jana
Yang Wang
John Urbanic

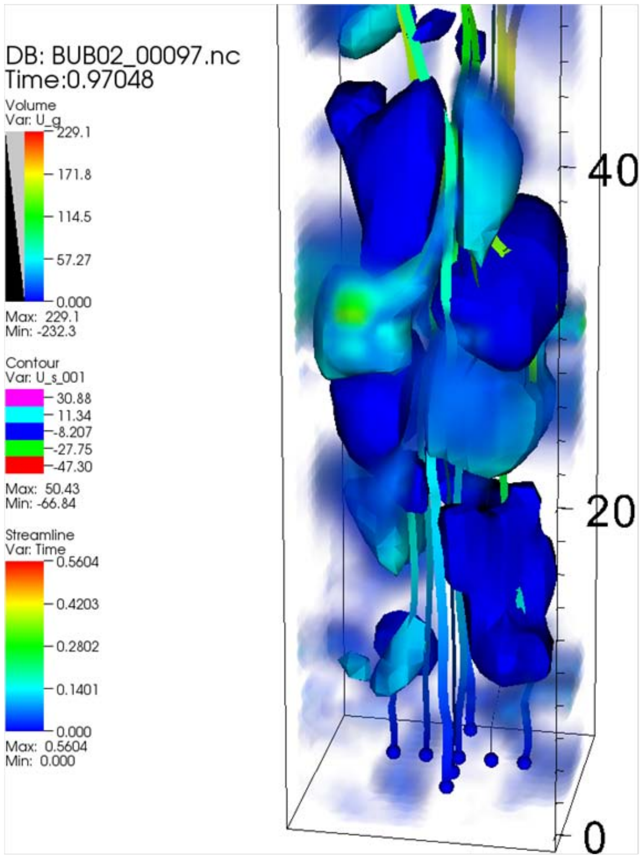


Figure 1. Image of a 3D Cartesian dataset showing gas velocity streamlines as ribbons.

How we came to this point.

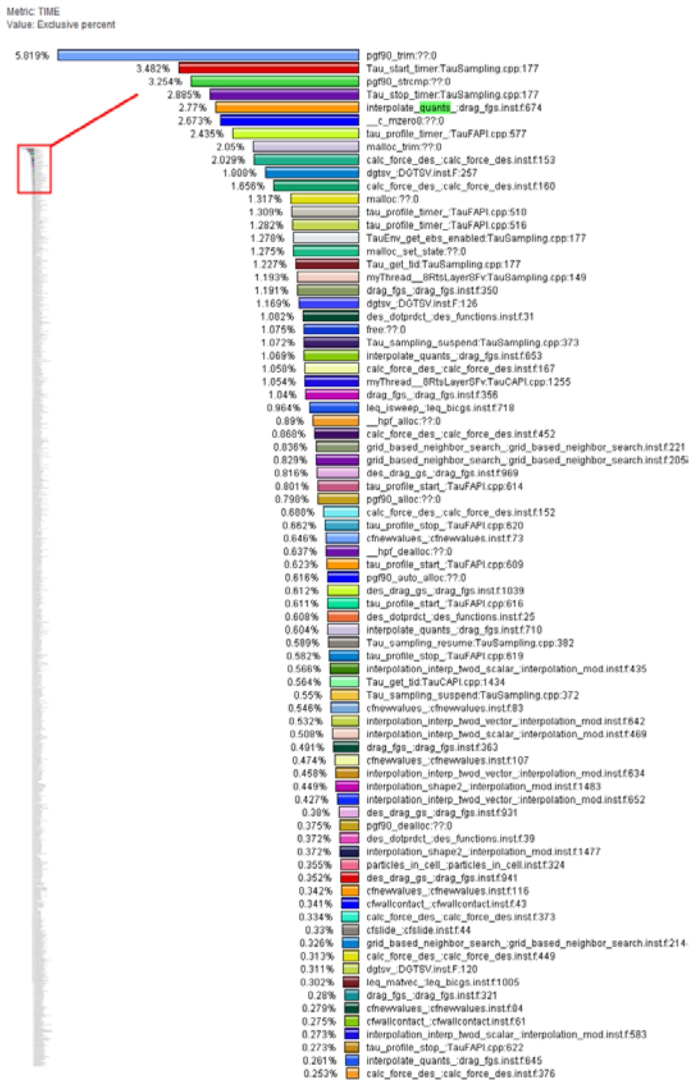
PSC has been working with NETL on MFIX for some time.

Last year we profiled MFIX in the course of some of that work.

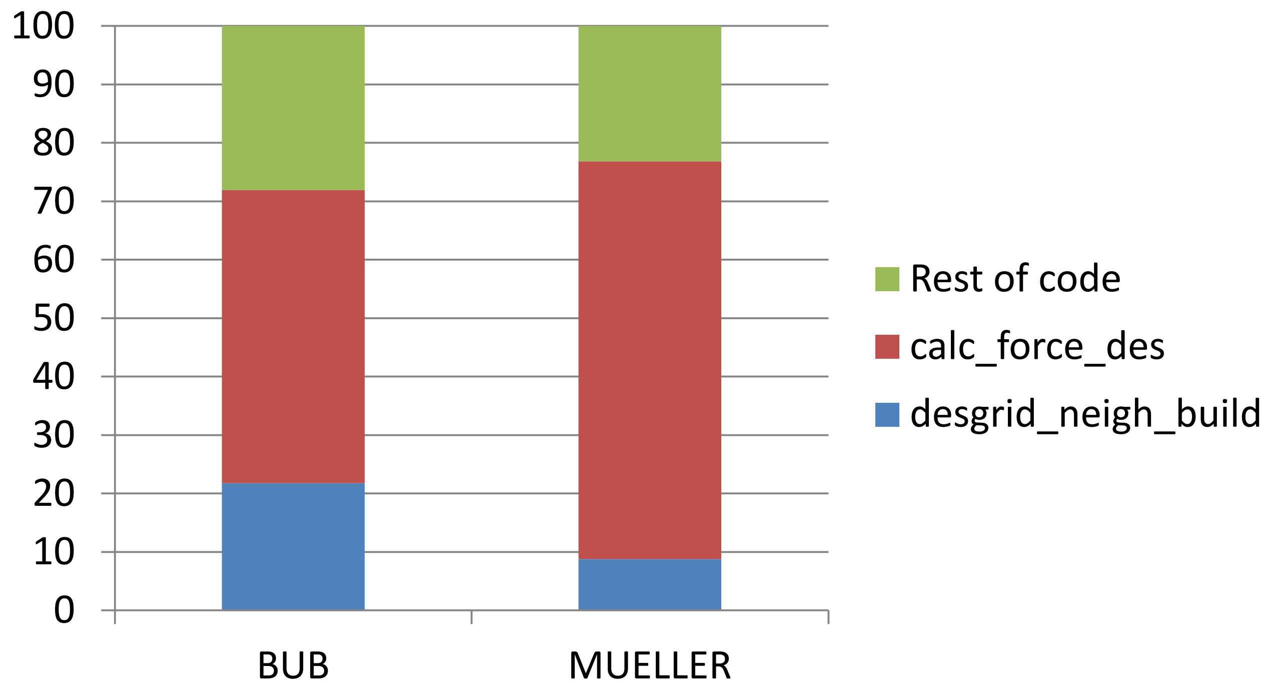
We used Tau, a sophisticated performance profiler with countless perspectives on the performance data.

It also had considerable overhead and “interpretation” issues.

Nevertheless, we were able to capture many useful visualizations of the work distribution under various conditions (datasets).



An obvious hotspot!



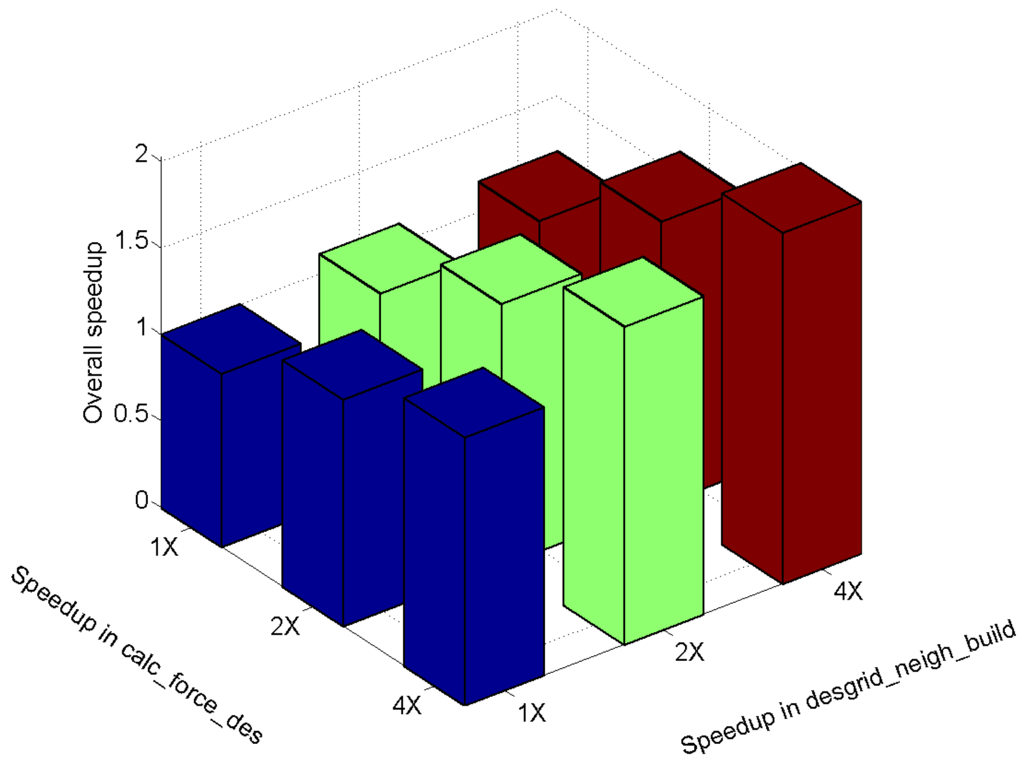
For our current work, we reproduced these timings with the latest code, a new benchmark dataset and manual timers. We obtained similar results.

Two algorithmic sections of the code dominate the runtime:

- Neighbor Search
- Force Calculation

Potential?

If we could somehow accelerate these two routines appreciably, we would get signigicant speedups for the overall code. Amdahl's Law at work.



A Reasonable Approach

After some initial research it seemed likely that these routines would be amenable to GPGPU type acceleration.

We concurrently looked into possible algorithms as well as the technical specifics of how to implement them.

We had a high degree of confidence that we could implement both algorithms in low-level GPU programming language (CUDA) that could deliver a performance improvement, perhaps a very substantial one.

However, as we were beginning this project, a new approach to GPU programming was finally reaching usability: the OpenACC standard. This has the potential to greatly mitigate our disturbance to the standard MFIX distribution.

Simplest Example: SAXPY on CPU

Single Precision Alpha X Plus Y (SAXPY) is a simple, common operation. It is part of BLAS.

Fortran:

```
      subroutine saxpy (A,X,Y,N)
        real(4) :: A, X(N), Y(N)
        integer :: N, I
        do i = 1,N
         X(i) = A * X(i) + Y(i)
        enddo
       end subroutine
```

C:

```
      void saxpy( float a, float* x, float* y, int n ){
        int i;
        for( i = 0; i < n; ++i ){
         x[i] = a*x[i] + y[i];
        }
       }
```

CUDA C SAXPY Code

```
__global__ void saxpy_kernel( float a, float* x, float* y, int n
   ){
  int i;
  i = blockIdx.x*blockDim.x + threadIdx.x;
  if( i <= n ) x[i] = a*x[i] + y[i];
}

void saxpy( float a, float* x, float* y, int n ){
  float *xd, *yd;
  cudaMalloc( (void**)&xd, n*sizeof(float) );
  cudaMalloc( (void**)&yd, n*sizeof(float) ); cudaMemcpy( xd, x,
   n*sizeof(float),
                    cudaMemcpyHostToDevice );
  cudaMemcpy( yd, y, n*sizeof(float),
                    cudaMemcpyHostToDevice );
  saxpy_kernel<<< (n+31)/32, 32 >>>( a, xd, yd, n );
  cudaMemcpy( x, xd, n*sizeof(float),
                    cudaMemcpyDeviceToHost );
  cudaFree( xd ); cudaFree( yd );
}
```

CUDA Fortran SAXPY Code

```
module kmod
 use cudafor
contains
 attributes(global) subroutine saxpy_kernel(A,X,Y,N)
  real(4), device :: A, X(N), Y(N)
  integer, value :: N
  integer :: i
  i = (blockidx%x-1)*blockdim%x + threadidx%x
  if( i <= N ) X(i) = A*X(i) + Y(i)
 end subroutine
end module

 subroutine saxpy( A, X, Y, N )
  use kmod
  real(4) :: A, X(N), Y(N)
  integer :: N
  real(4), device, allocatable, dimension(:):: &
                Xd, Yd
  allocate( Xd(N), Yd(N) )
  Xd = X(1:N)
  Yd = Y(1:N)
  call saxpy_kernel<<<(N+31)/32,32>>>(A, Xd, Yd, N)
  X(1:N) = Xd
  deallocate( Xd, Yd )
 end subroutine
```

What a mess!

- We totally broke the structure of the old code.
  - –Do you think any non-CUDA programmers have a prayer dealing with this code?
- We have separate sections for the host code, and the GPU code.
  - –Certainly not ANSI standard.
- Where did these “32’s” and other mystery variables come from?
- Can we avoid doing this to the MFIX distribution?

Answer: OpenACC

Fortran:

```
       subroutine saxpy (A,X,Y,N)
         real(4) :: A, X(N), Y(N)
         integer :: N, I
       !$acc parallel
         do i = 1,N
          X(i) = A * X(i) + Y(i)
         enddo
       !$acc end parallel
        end subroutine
```

C:

```
       void saxpy( float a, float* x, float* y, int n ){
         int i;
         #pragma acc parallel
         for( i = 0; i < n; ++i ){
          x[i] = a*x[i] + y[i];
         }
        }
```

Maintenance Pros of OpenACC

- **No separate code path (ideally), whereas CUDA is a “fork”.**
  We might not be able to achieve that on our first pass here as we want to leverage an existing CUDA sort library.

- **No GPU platform specifics by generation/vendor.**
  AMD/Intel not on board yet, but will be by *OpenMP 4.0**

- **Can learn OpenACC in a day, CUDA takes at least several and is a moving target.**

*** OpenMP programmers, note how similar this paradigm is to standard multi-core OpenMP. This is why OpenACC is due for integration into the OpenMP 4.0 standard, likely in 2013.**

Risks of OpenACC

- **Might not get full performance of CUDA.**
  - **Worst case, we fall back on CUDA.**

- **Currently relying on beta compilers.**
  - **Fortunately, we have a very close relationship with PGI on this product.**
  - **Recently, we have become the leading edge site with nVidia on this.**

- **Not all of the specification is implemented.**
  - **We think we have all the pieces for our algorithms now.**
  - **Full implementation may take until 4Q12.**

Current Plan in a Nutshell

1. ~~Profile code and find hotspots~~
2. Create MFIX data structure skeleton to test prototypes
3. Find suitable replacement algorithms
4. Code replacement algorithms
5. Tune prototype code
6. Swap into MFIX

Testbed for Neighbor Search

- A light weight framework for implementing and testing a new neighbor list calculation method
- Consisting of a main driver (testNeighborSearch.f90) and supporting routines from MFIX
- Three functions of the testbed:
  - Initializing the system geometry and other simulation parameters
  - Generating the particle positions
  - Calling a neighbor search routine to calculate the neighbor list. Currently, it calls NSQUARE(), QUARDTREE, or OCTREE(), implemented in MFIX.
- Steps forward:
  - Implementing CELLLIST(), a new neighbor search routine
  - Implementing openACC in CELLLIST()
  - Placing CELLLIST(), capable of running on GPU, into MFIX

Current Plan in a Nutshell

1. ~~Profile code and find hotspots~~
2. ~~Create MFIX data structure skeleton to test prototypes~~
3. Find suitable replacement algorithms
4. Code replacement algorithms
5. Tune prototype code
6. Swap into MFIX

Combined Neighbor Search Algorithm

- The algorithm combines the following two techniques
  - Neighbor lists approach: a neighbor list is built for each atom for an extended cut off radius $r_s = r_c + \gamma$
  - Link-cell approach: atoms are binned into 3d cells of side length $d = r_c$
- The atoms are binned only every few timesteps for the purpose of forming neighbor lists
  - Significant time savings compared to other approaches
  - Bin size $d \sim 0.5r_s$
- A GPU implementation has been built into LAMMPS, a parallel molecular dynamics simulation package developed in Sandia National Laboratory
- The radix sort routine in nVIDIA CUDPP library is used for sorting the atom and cell index arrays

Brown et al, "Implementing Molecular Dynamics on Hybrid High Performance Computers – Short Range Forces," *Computer Physics Communications*, **183**, 449 (2012)

Follow-up

- We hope the standard distribution sports these changes by year end.
- If you find this OpenACC approach interesting, we hope to do a hands-on workshop with PGI and nVidia at SC12. Please join us then (if not before).