Python for Development of OpenMP and CUDA Kernels for Multidimensional Data

Zane W. Bell[1], Greg G. Davidson[2], Ed D'Azevedo[3], Thomas M. Evans[2], Wayne Joubert[4], John K. Munro, Jr.[5], Dilip R. Patlolla[5] and Bogdan Vacaliuc[5]

[1] Nuclear Material Detection & Characterization/NSTD/ORNL

[2] Radiation Transport/RNSD/ORNL

[3] Computational Mathematics/CSMD/ORNL

[4] Scientific Computing/CCSD/ORNL

[5] Measurement Science and Systems Engineering/EESD/ORNL

2011 Symposium on Application Accelerators in HPC

20 July 2011

Overview

- Use Python environment
  - Problem setup, data structure manipulation, file I/O
  - The “architecture of the computation”

- Implement optimal computation kernels in C++, Fortran, CUDA or 3rd Party APIs
  - Leverage experts and existing code subroutines
  - The “details” of the computation

“Raising the level of programming should be the single most important goal for language designers, as it has the greatest effect on programmer productivity.”

J. Osterhout [14]

Boltzmann Transport Equation

The Boltzmann transport equation for the special case of one dimensional, spherical symmetry, discrete ordinates, time-independent transport is

$$\frac{\mu}{r^2}\frac{\partial}{\partial r}[r^2\psi(r,\mu,E)] + \frac{1}{r}\frac{\partial}{\partial \mu}[(1-\mu^2)\psi(r,\mu,E)] + \sigma(r,E)\psi(r,\mu,E) =$$

$$\int_E 2\pi \int_{-1}^{1} \sigma_s(r,\mu\cdot\mu',E' \rightarrow E)\psi(r,\mu',E')d\mu' dE' + q(r,\mu,E)$$

Where

ψ Is the radiation intensity (flux) at position *r*, with energy *E* moving in μ

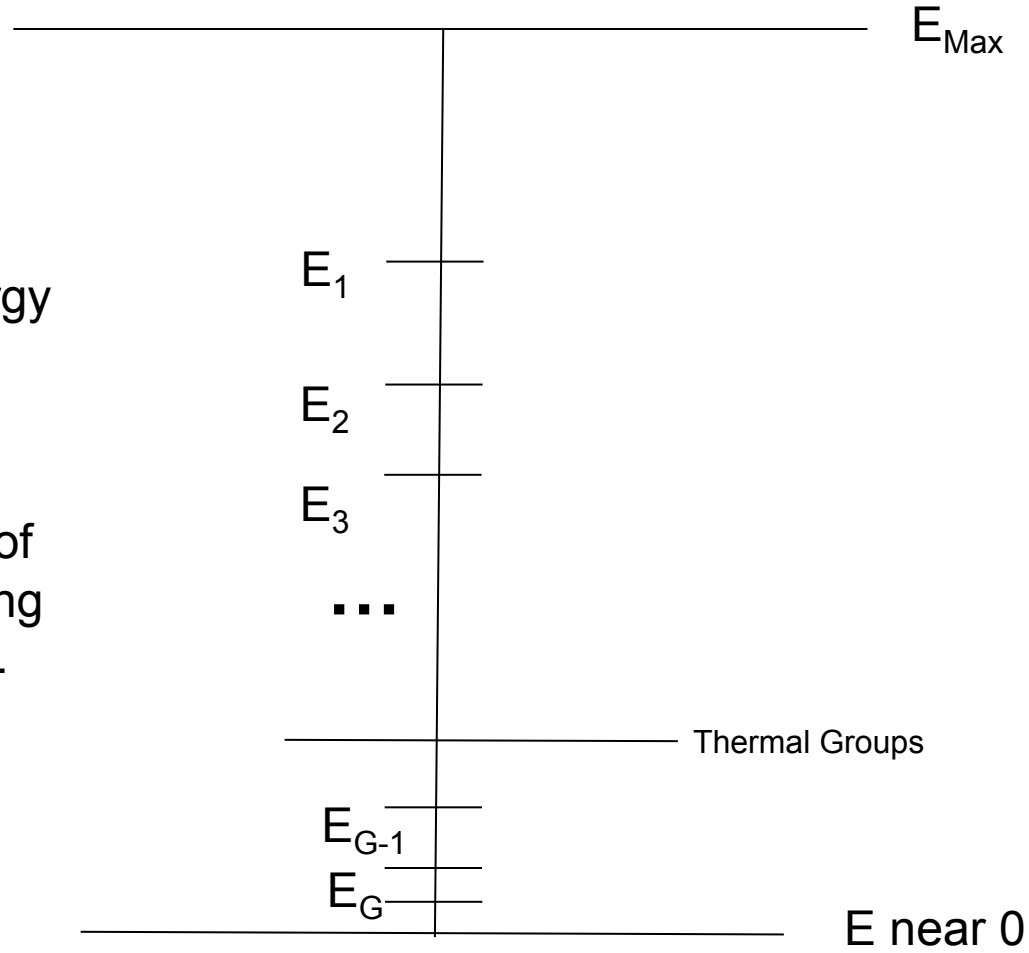
σ and σ_s are the total and scattering cross-sections

q is the external source particle density

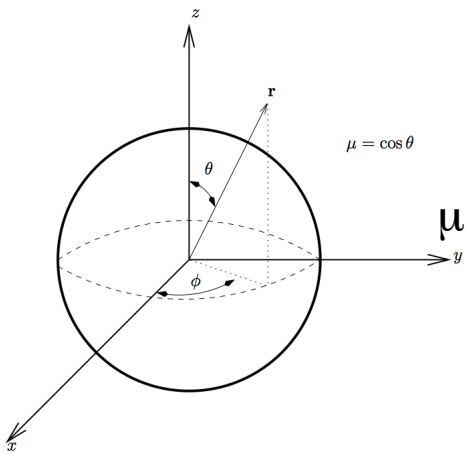
To solve numerically, we discretize in energy, angle and radial terms.

Energy Discretization

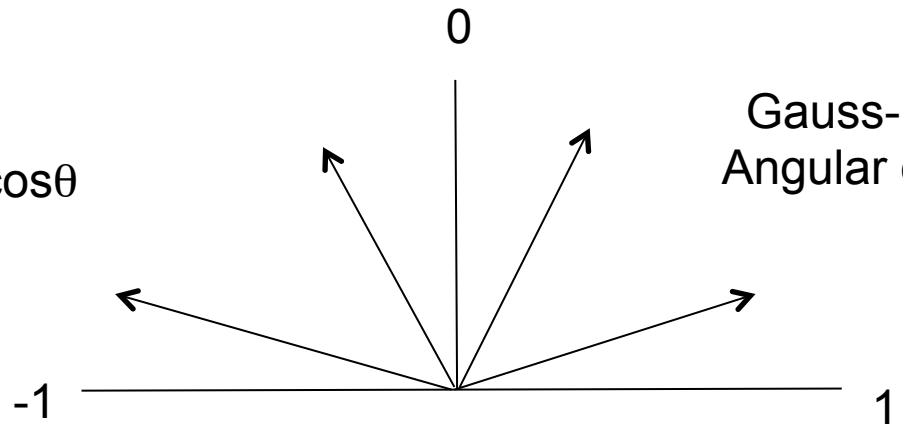
- Choose number of energy groups (*G*) and E_{Max} to correspond to the resolution of interest
- Energy groups may be of different sizes, depending on resolution of interest.



Angular and Radial Discretization



$\mu = \cos\theta$

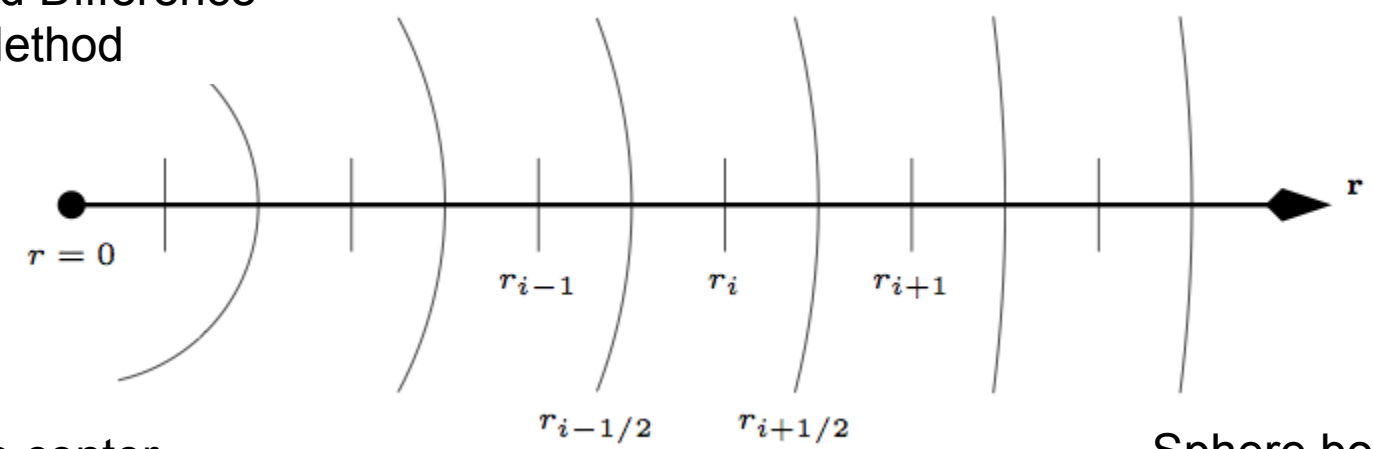


Gauss-Legendre
Angular quadrature

Toward sphere center

Toward sphere boundary

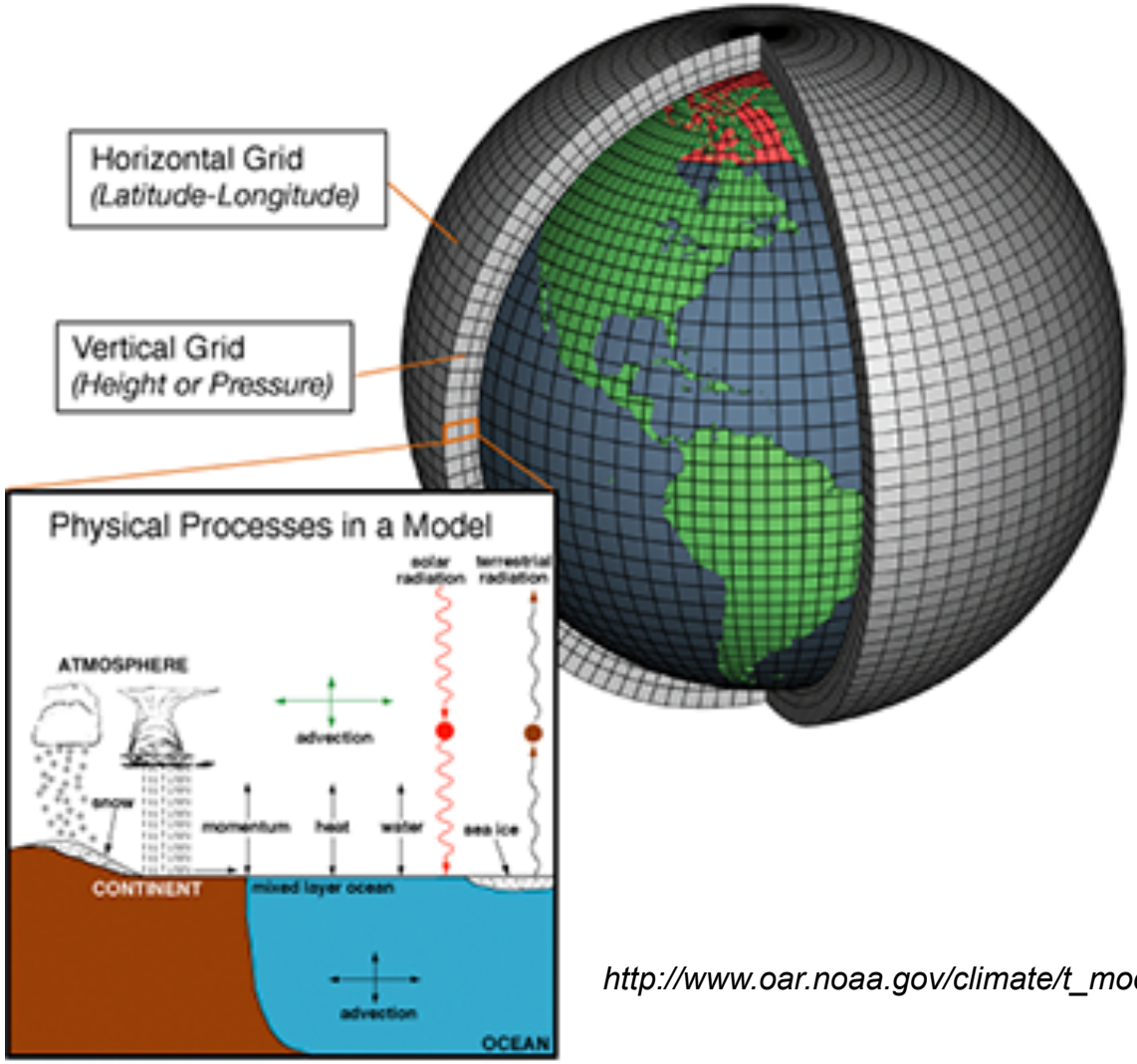
Diamond Difference
Method



Sphere center

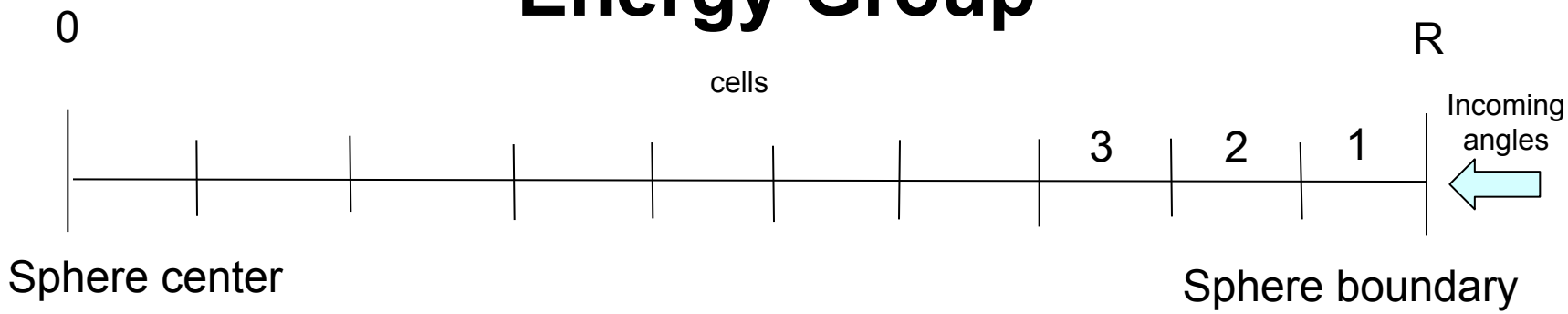
Sphere boundary

Angular and Radial Discretization

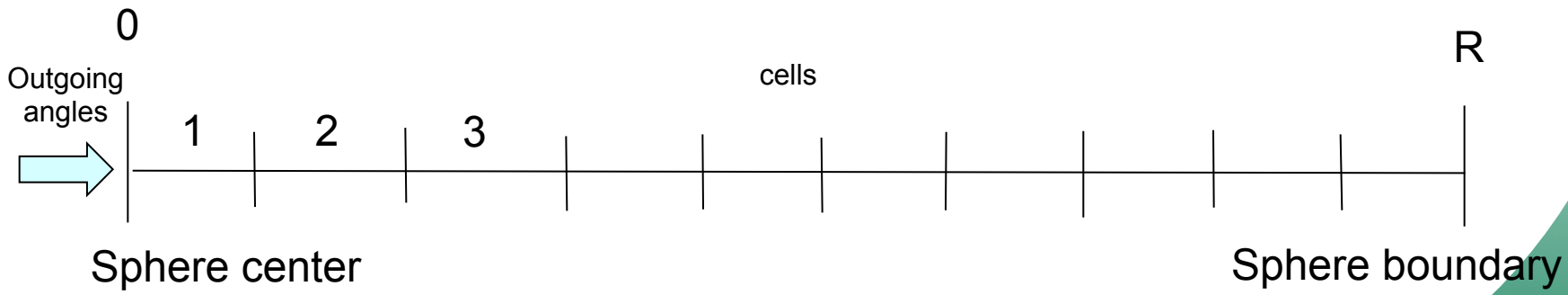


http://www.oar.noaa.gov/climate/t_modeling.html

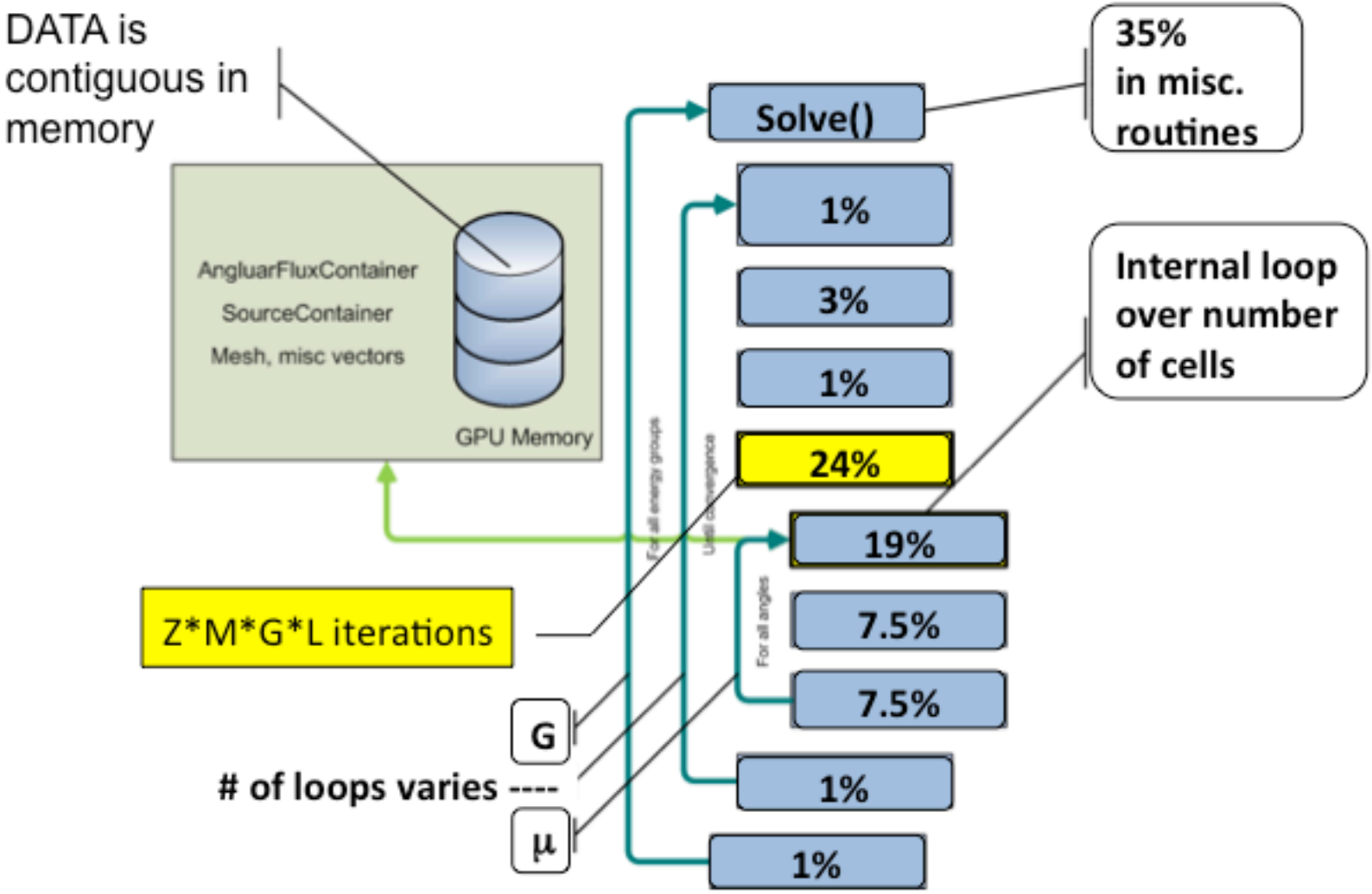
“Sweep” radial cells within Each Energy Group



- A transport “sweep” is the process of solving the diamond difference, space-angle S_N equations
  - A wavefront solution in which the value of each cell depends on the flux entering in the “upwind” direction.



Algorithm Structure and Profile



Python Reference Implementation

(prob1.py)

```
// read-only:  a_sxs[G][G][L+1]
// read-only:  a_ofm[G][Z][L+1]
// read-only:  a_ext[G]
// read-only:  a_mu[M]
// write-only: r_src[G][Z][M]
```

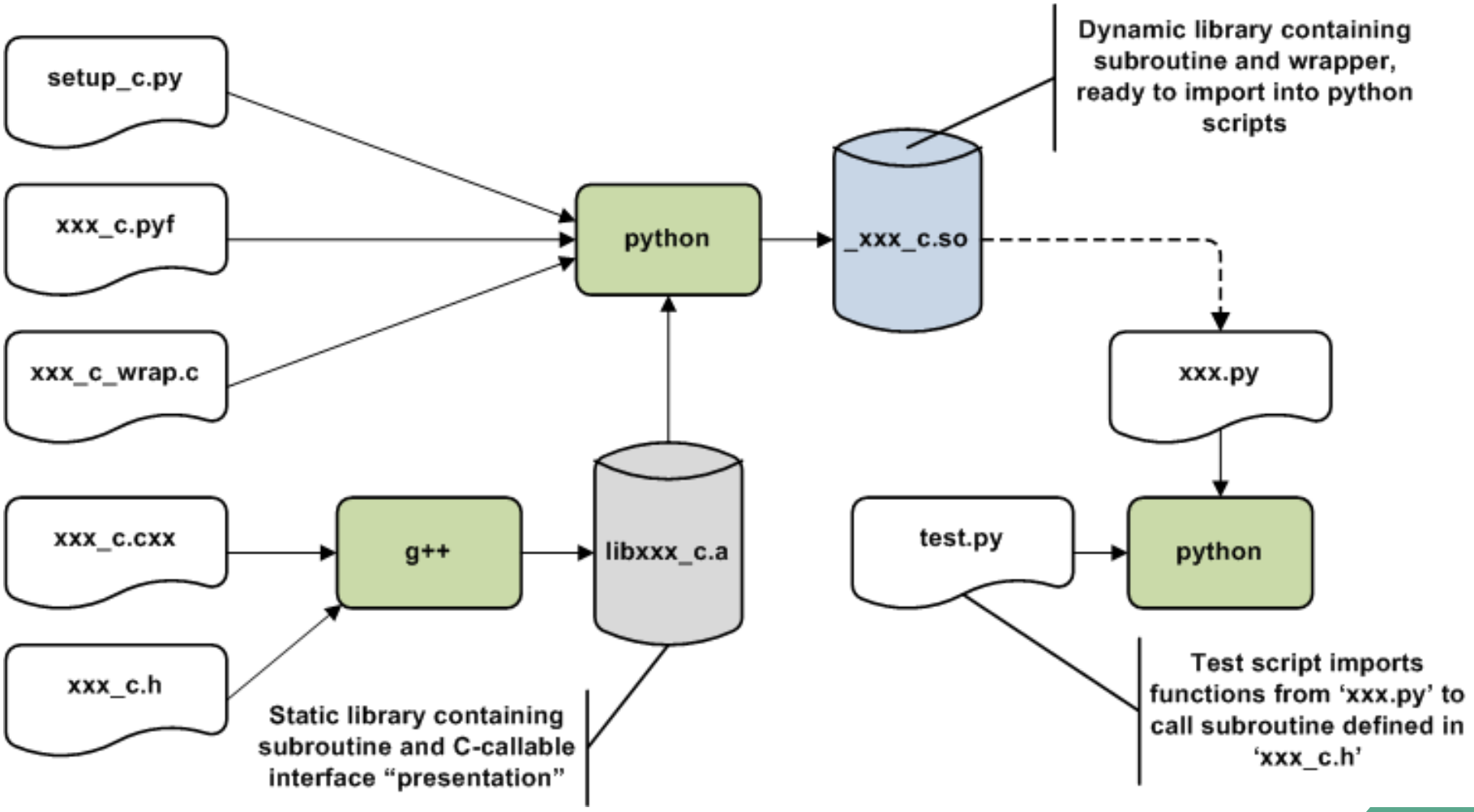
```
def prob1(Z,M,G,L,a_sxs,a_ofm,a_ext,a_mu):
 r_src = zeros([G,Z,M]).astype(a_ext.dtype)
 for z in range(0,Z):
  for m in range(0,M):
   ss = 0.0
   for g in reversed(range(0,G)):
    for el in range(0,L+1):    # NB: [0,L+1)
     v  = plgndr(el,a_mu[m])
     ss = ss + (2*el+1)/(4*(PI)) * a_sxs[G-1,g,el] * v * a_ofm[g,z,el]
   r_src[G-1,z,m] = ss + a_ext[G-1,0]/(4*(PI))
 return r_src
```

C++ Template Implementation

(prob1_c.h)

```
namespace ce
{
   template <typename T>
   void prob1(const int Z,
              const int M,
              const int G,
              const int L,
              const T *a_sxs, // read-only:  a_szx[G][G][L+1]
              const T *a_ofm, // read-only:  a_ofm[G][Z][L+1]
              const T *a_ext, // read-only:  a_ext[G]
              const T *a_mu,  // read-only:  a_mu[M]
              T *r_src)       // write-only: r_src[G][Z][M]
   {
     const T PI = 3.1415926535827;
     T ss, v;
     int z,m,g,l;
     uint32_t i_src,i_sxs,i_ofm;

     #pragma omp parallel for shared(a_sxs,a_ofm,a_ext,a_mu,r_src) \
                              private(ss,v,z,m,i_src,g,l,i_sxs,i_ofm)
     for(z = 0; z < Z; ++z) {
       for(m = 0; m < M; ++m) {
         ss = 0.0;
         i_src = (G-1)*(Z*M) + z*M + m;              // r_src[G-1][z][m]
         for(g = G-1; g >= 0; --g ) {
           for(el = 0; el <= L; ++el) {              // NB: [0,L+1)
             v = gsl::plgndr<T>(el, a_mu[m]);
             i_sxs = (G-1)*(G*(L+1)) + g*(L+1) + el; // a_sxs[G-1][g][el]
             i_ofm = g*(Z*(L+1)) + z*(L+1) + el;     // a_ofm[g][z][el]
             ss += ( (2.0*el + 1.0) / (4.0*PI) ) * a_sxs[i_sxs] * v * a_ofm[i_ofm];
           }
         }
         r_src[i_src] = ss + a_ext[G-1] / (4.0*PI);
       }
     }

   };
}
```

Flow for C++ Wrapper



Python F2PY Interface Declaration

(prob1_c.pyf)

```
// read-only:  a_sxs[G][G][L+1]
// read-only:  a_ofm[G][Z][L+1]
// read-only:  a_ext[G]
// read-only:  a_mu[M]
// write-only: r_src[G][Z][M]
```

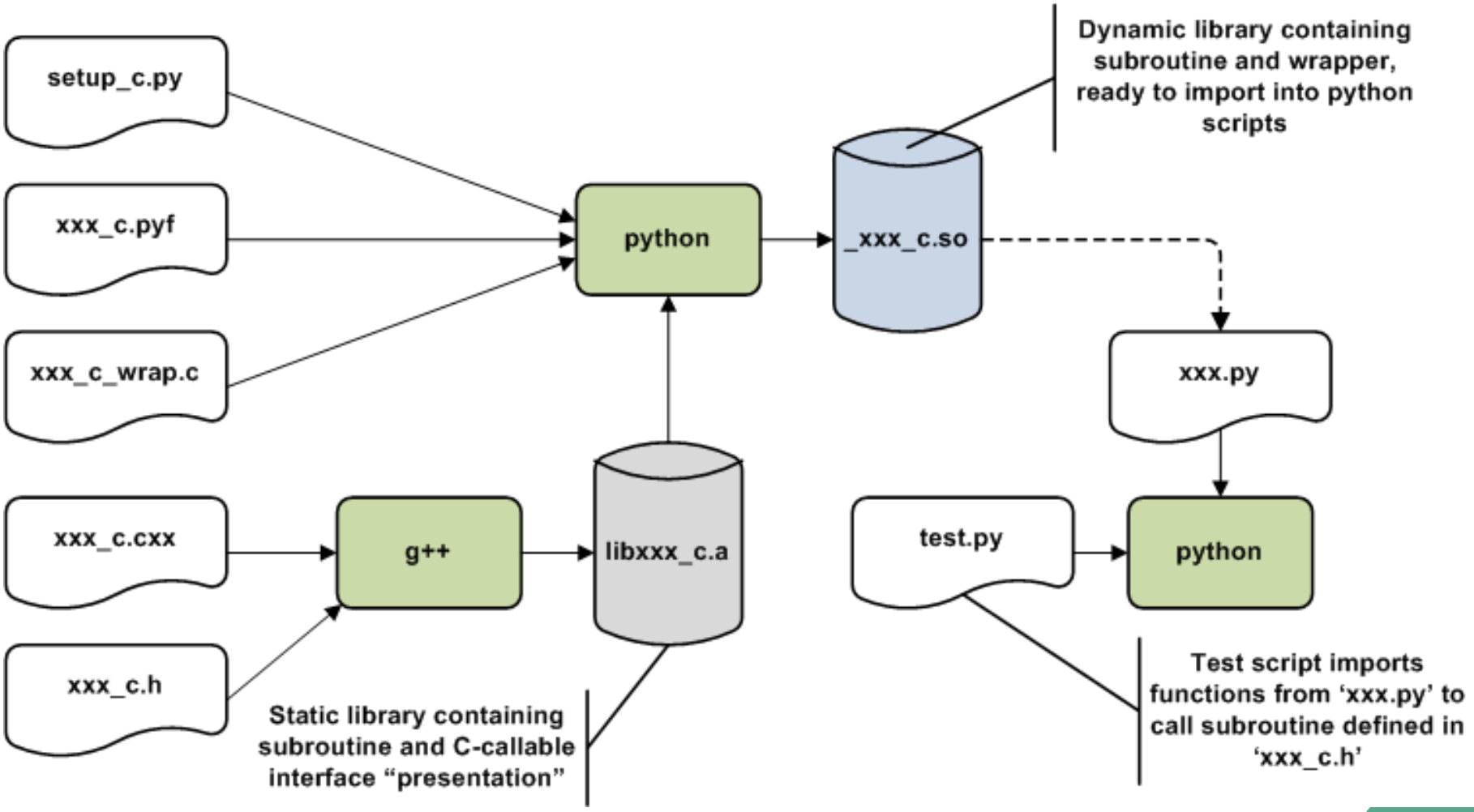
```
! -*- f90 -*-
! File prob1_c.pyf
python module _prob1_c
interface
  subroutine prob1_dp(z,m,g,l,sxs,ofm,ext,mu,src)
    intent(c) prob1_dp              ! is a C function
    intent(c)                       ! all arguments are
                                    ! considered as C based
    integer intent(in) :: z
    integer intent(in) :: m
    integer intent(in) :: g
    integer intent(in) :: l
    real*8  intent(in),dimension(g,g,l+1),depend(g,l)     :: sxs(g,g,l+1)
    real*8  intent(in),dimension(g,z,l+1),depend(g,z,l)   :: ofm(g,z,l+1)
    real*8  intent(in),dimension(g),depend(g)             :: ext(g)
    real*8  intent(in),dimension(m),depend(m)             :: mu(m)
    real*8  intent(out),dimension(g,z,m),depend(g,z,m)    :: src(g,z,m)
  end subroutine prob1_dp
```

Python C++/F2PY Interface Building

(setup_c.py and makefile)

```
# File setup_c.py
def configuration(parent_package='',top_path=None):
    from numpy.distutils.misc_util import Configuration
    config = Configuration('',parent_package,top_path)

    config.add_library(name='prob1_c', sources=['prob1_c.cxx'])
    config.add_extension('_prob1_c',
                         sources = ['prob1_c.pyf','prob1_c_wrap.c'],
                         libraries = ['prob1_c'])
    return config
if __name__ == "__main__":
    from numpy.distutils.core import setup
    setup(**configuration(top_path='').todict())


# build OpenMP-versions
omp:
 @( export ARCHFLAGS=$(ARCHFLAGS) ; \
    export CPPFLAGS="-fopenmp $(TUNE)" ; \
    export LDFLAGS="-lgomp" ; \
    python setup_c.py build_src build_ext --inplace )
```

Flow for C++ Wrapper (again)



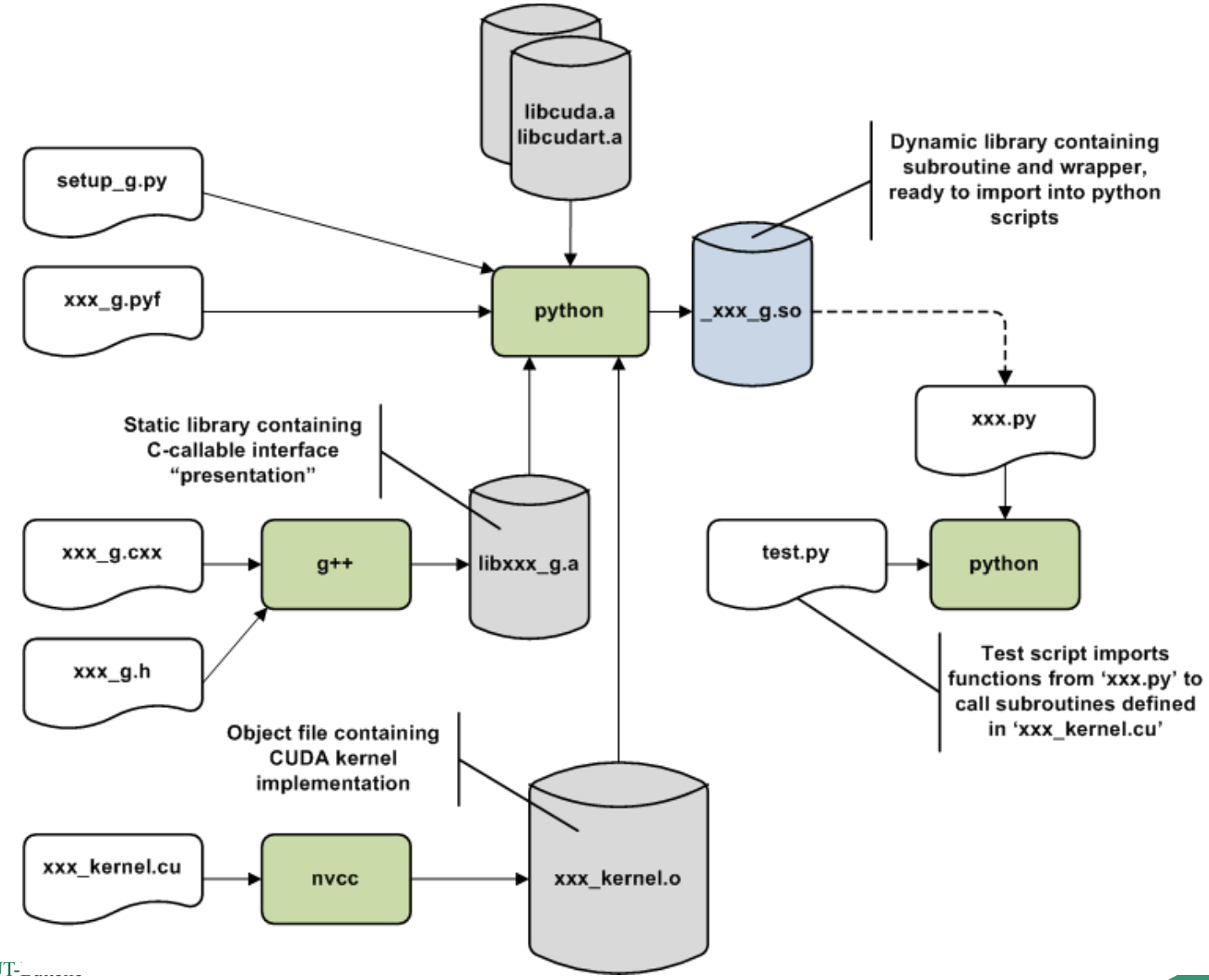
Python Call C++ Kernel

(prob1.py)

```
# interface C-code via F2PY
def prob1_c_f2py(Z,M,G,L,a_sxs,a_ofm,a_ext,a_mu):
 import _prob1_c as c_f2py
 if len(Z.shape) > 1:
  Z = Z[0,0]
 if len(M.shape) > 1:
  M = M[0,0]
 if len(G.shape) > 1:
  G = G[0,0]
 if len(L.shape) > 1:
  L = L[0,0]
 r_src = zeros([G,Z,M]).astype(a_ext.dtype)
 if a_ext.dtype == "float64":
  r_src = c_f2py.prob1_dp(Z,M,G,L,a_sxs,a_ofm,a_ext,a_mu)
 else:
  r_src = c_f2py.prob1_sp(Z,M,G,L,a_sxs,a_ofm,a_ext,a_mu)
 return r_src
```

```
// read-only:  a_sxs[G][G][L+1]
// read-only:  a_ofm[G][Z][L+1]
// read-only:  a_ext[G]
// read-only:  a_mu[M]
// write-only: r_src[G][Z][M]
```

Flow for CUDA Wrapper



Python GPU/F2PY Interface Building

(setup_g.py)

```
# File setup_g.py
# See also:
#   http://www.scipy.org/Dynetrekk/f2py_OpenMP_draft
#
def configuration(parent_package='',top_path=None):
    from numpy.distutils.misc_util import Configuration
    config = Configuration('',parent_package,top_path)

    config.add_library(name='prob1_g',
                            sources=['prob1_g.cxx'])
    config.add_extension('_prob1_g',
                        sources = ['prob1_g.pyf','prob1_c_wrap.c'],
                              extra_objects = ['prob1_kernel.o'],
                        libraries = ['prob1_g','cuda','cudart'])
    return config
if __name__ == "__main__":
    from numpy.distutils.core import setup
    setup(**configuration(top_path='').todict())
```

Python GPU/F2PY Interface Building

(makefile)

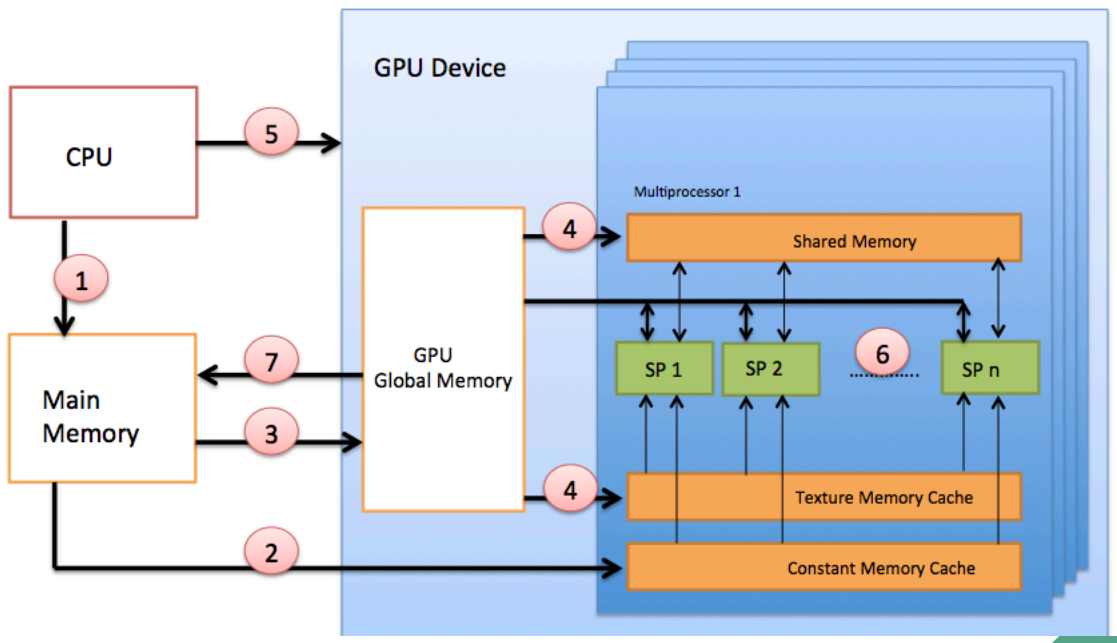
```
# build GPU-versions
$(MOD)_kernel.o: $(MOD)_kernel.h $(MOD)_kernel.cu
 nvcc $(NVCC_CU_FLAGS) $(INCLUDES) -c $(MOD)_kernel.cu

_$(MOD)_g.so: $(MOD)_kernel.o $(MOD)_g.h $(MOD)_g.cxx $(MOD)_c_wrap.c
              $(MOD)_g.pyf setup_g.py
 ( export ARCHFLAGS=$(ARCHFLAGS) ; \
   export CPPFLAGS="-fopenmp $(TUNE)" ; \
   export LDFLAGS="-L$(CLIB) -lgomp" ; \
   python setup_g.py build_ext --inplace )

gpu: _$(MOD)_g.so
```

“problem set #1”

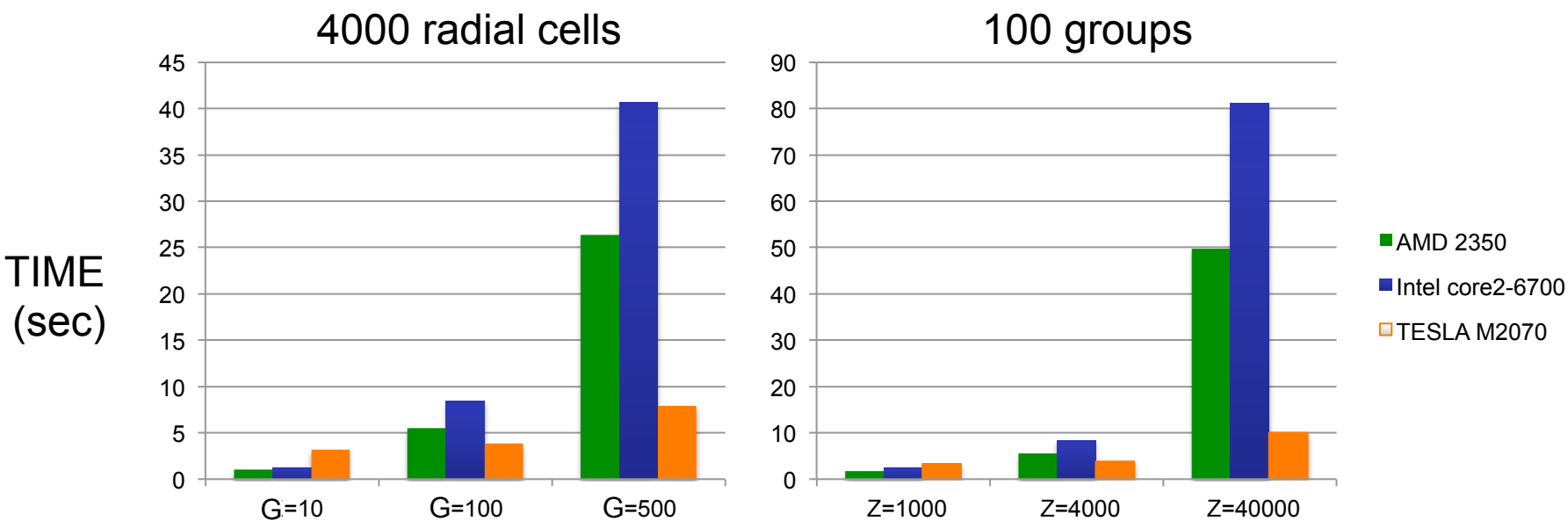
- Loop over all radial cells (Z)
  - Loop over all angles (M, typically 8)
    - Loop over all energy groups (G)
      - Integrate Legendre expansion for this angluar moment, accumulate the source term (L ~ 24th order)
  - Update the radial cell source term

Z*M*G*L integrations

1. Compute Legendre table in CPU memory
2. Copy Legendre table lookup to constant memory on GPU
3. Copy solver state to GPU (Unit Test Only)
4. Load Shared Memory
5. Execute
6. SP’s work
7. Copy result (angular flux) to CPU memory (Unit Test Only)



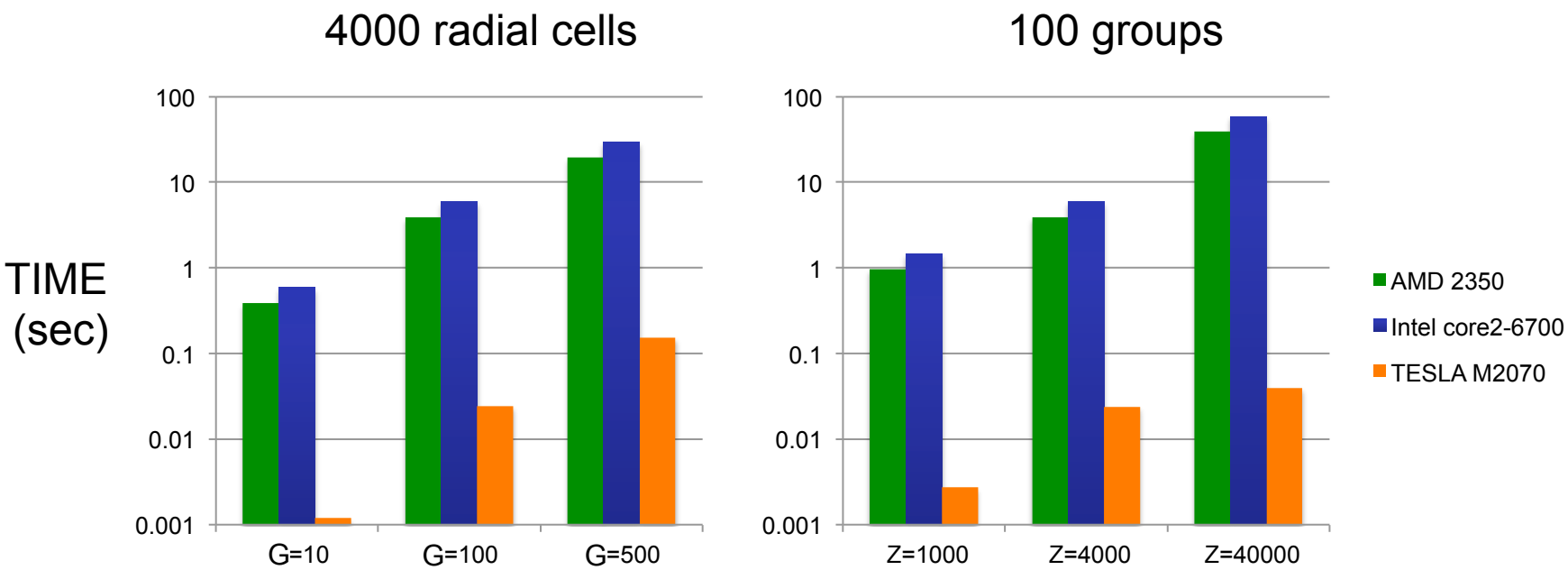
Runtime Comparison (with I/O overhead)



"prob1", 24th order Legendre expansion, 8 angles

- Verified matching results for single-precision, double-precision ~ 1e-6
- Fermi implements accelerator-model speedup of 1.3x to 6.2x
  - accounting for the I/O to and from CPU memory
  - versus 2 and 4 core CPUs

Runtime Comparison (kernel only)



"prob1", 24th order Legendre expansion, 8 angles

- NOTE: logarithmic scale
- Kernel-only timing shows 65x to 115x speedup (vs. 2-core CPU)
  - OK, because our final code has all data resident on the GPU memory
- Significant performance differences between experimental systems

Performance Model

$$LOAD = (Z \times M \times L)(G^2 + G),$$

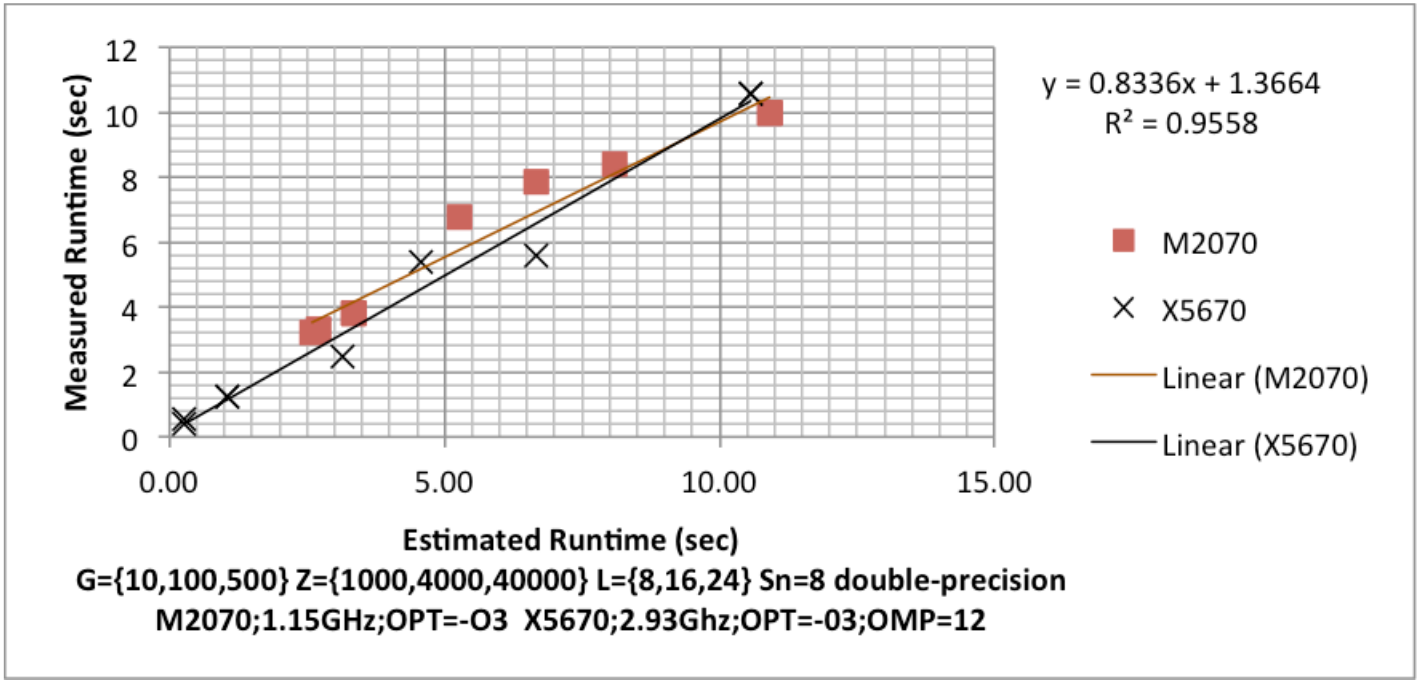
$$FLOPS = (Z \times M) \times \left({}^{11}/_{2}\,(L^2 + L)(G^2 + G) + 2\right),$$

$$STORE = (Z \times M).$$

$$T_{Estimated} = \frac{(LOAD + STORE) \times P_{bits} \times P_{mem}}{M_{clk} \times M_{width}} + \frac{FLOPS \times P_{fpu}}{P_{clk} \times P_{num}}$$

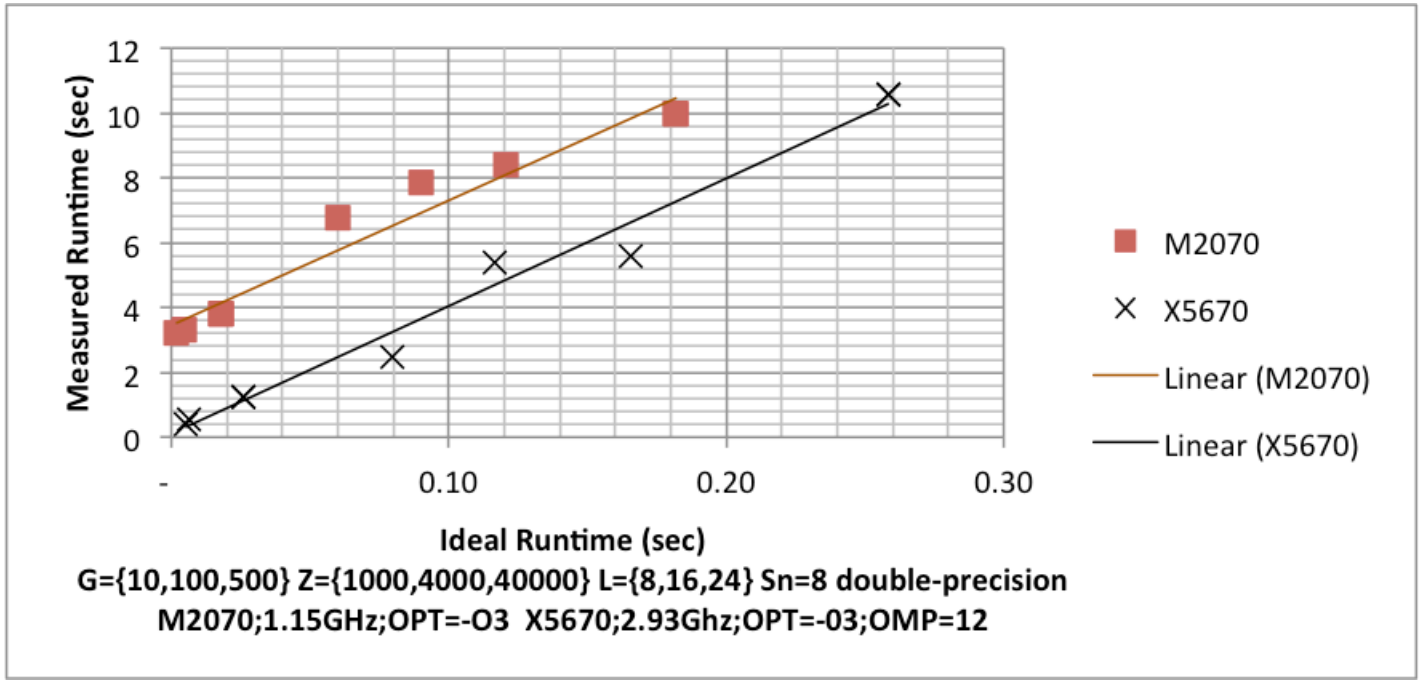
- P_{mem} and P_{fpu} are efficiency factors applied (simplified model)
- P_{bits} is 64 (IEEE-754 double-precision)
- Applied to both CPU and GPU (*naiive*)

CPU/GPU Comparison (with I/O overhead)



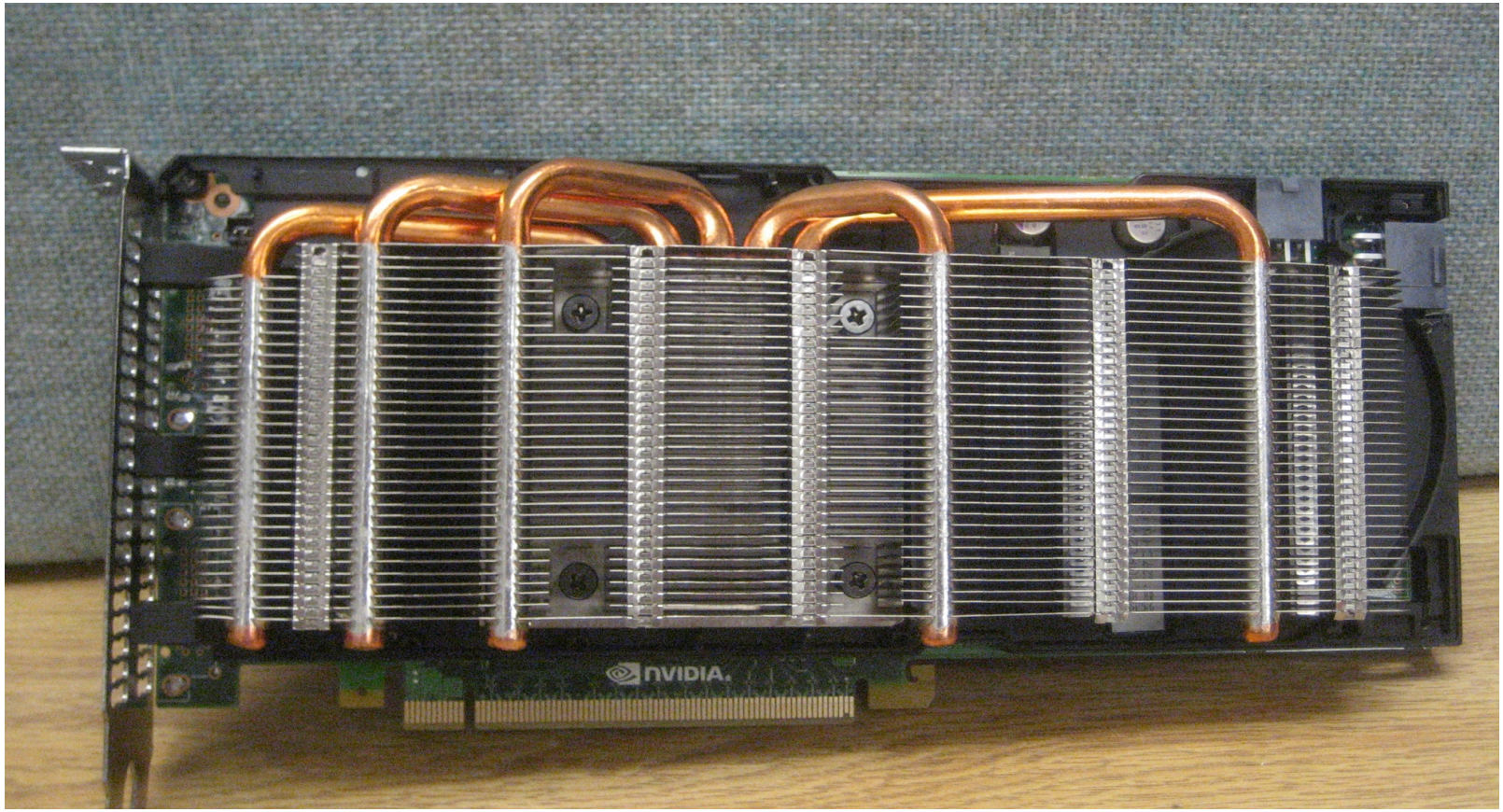
- Measured vs. Estimated Runtime (performance model)
  - M2070 factors in a 2.5 second application load delay (CUDA overhead)
- M2070 (448 cores, 225W) similar to Dual X5670 (12 cores, 190W)
  - M2070 $\{P_{mem}=46, P_{fpu}=50\ (2\%)\}$
  - X5670 $\{P_{mem}=12, P_{fpu}=58\ (1.7\%)\}$

CPU/GPU Comparison (with I/O overhead)

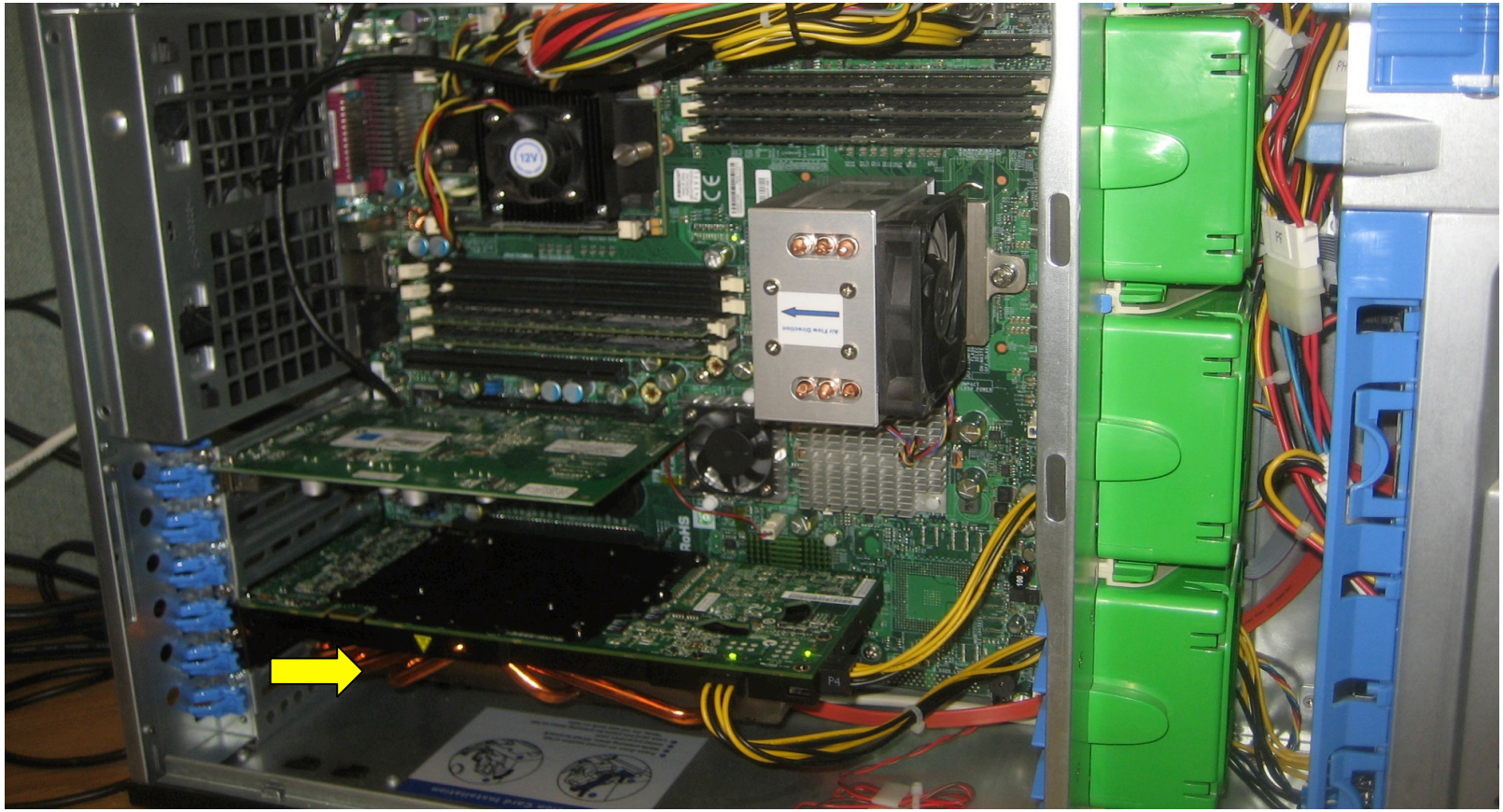


- Measured vs. Ideal Runtime (performance model)
  - P_{mem} and P_{fpu} set to 1
- M2070 (448 cores, 225W) similar to Dual X5670 (12 cores, 190W)
  - Keeping in mind that we are factoring the I/O overhead

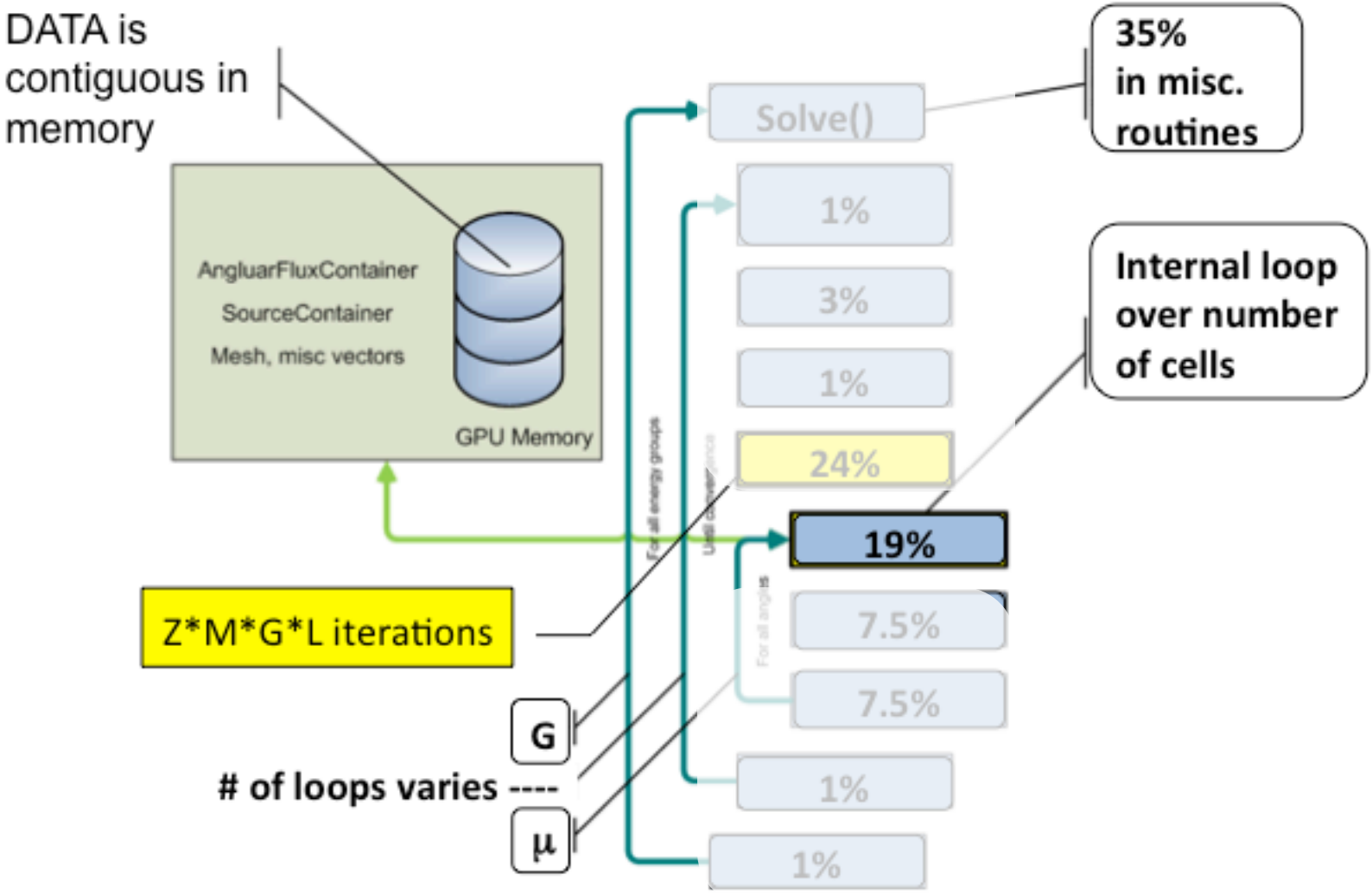
M2070 “Fermi” GPU



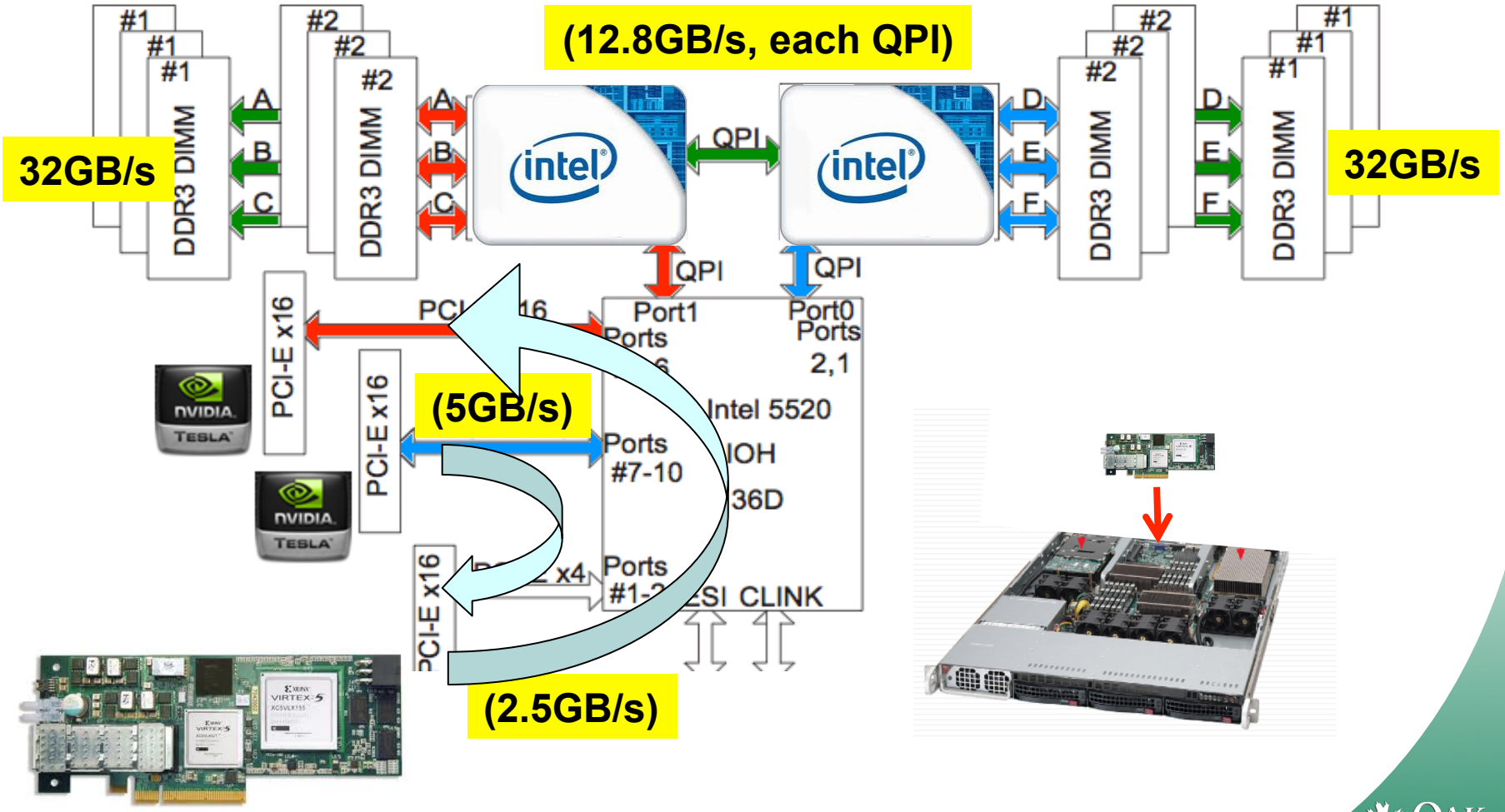
Multi-Core CPU, GPU, FPGA
“Exploratory System”



Next Task (#2) has Loop Dependency



Computational Engine: multi-core CPU with GPU and FPGA



Summary

- Use Python environment
  - Problem setup, data structure manipulation, file I/O
  - Use the wide array of available modules
  - Syntax similar to Matlab (*the scientists will like it*)

- Implement optimal computation kernels in C++, Fortran, CUDA or 3rd Party APIs
  - Leverage experts and existing code subroutines
  - Opportunities to use ASIC/Heterogenous Computation Devices (via API calls)

- All code referenced in this paper
  - http://info.ornl.gov/sites/publications/Files/Pub30033.tgz

"Raising the level of programming should be the single most important goal for language designers, as it has the greatest effect on programmer productivity."

J. Osterhout [14]


STS-135
The Final Mission
of the Space Shuttle
Watch landing LIVE
July 21 at 5:56 a.m. EDT
United States
Thank You