Scientific programming with graphics processing units

Eugene DePrince
Florida State University

Graphics processing units

3 of top 10 machines (top500.org) use NVIDIA hardware

#2: Titan (ORNL)
AMD + NVIDIA K20x

Also, does anyone listen to NPR??

Titan got a shout-out yesterday!!

Graphics processing units

3 of top 10 machines (top500.org) use NVIDIA hardware

#2: Titan (ORNL)
AMD + NVIDIA K20x

today, we get to play with two systems:

Blueridge:

NVIDIA Tesla K40m (Kepler) GPU
2880 CUDA cores
12 GB global memory
1.43 Tflops max double precision performance

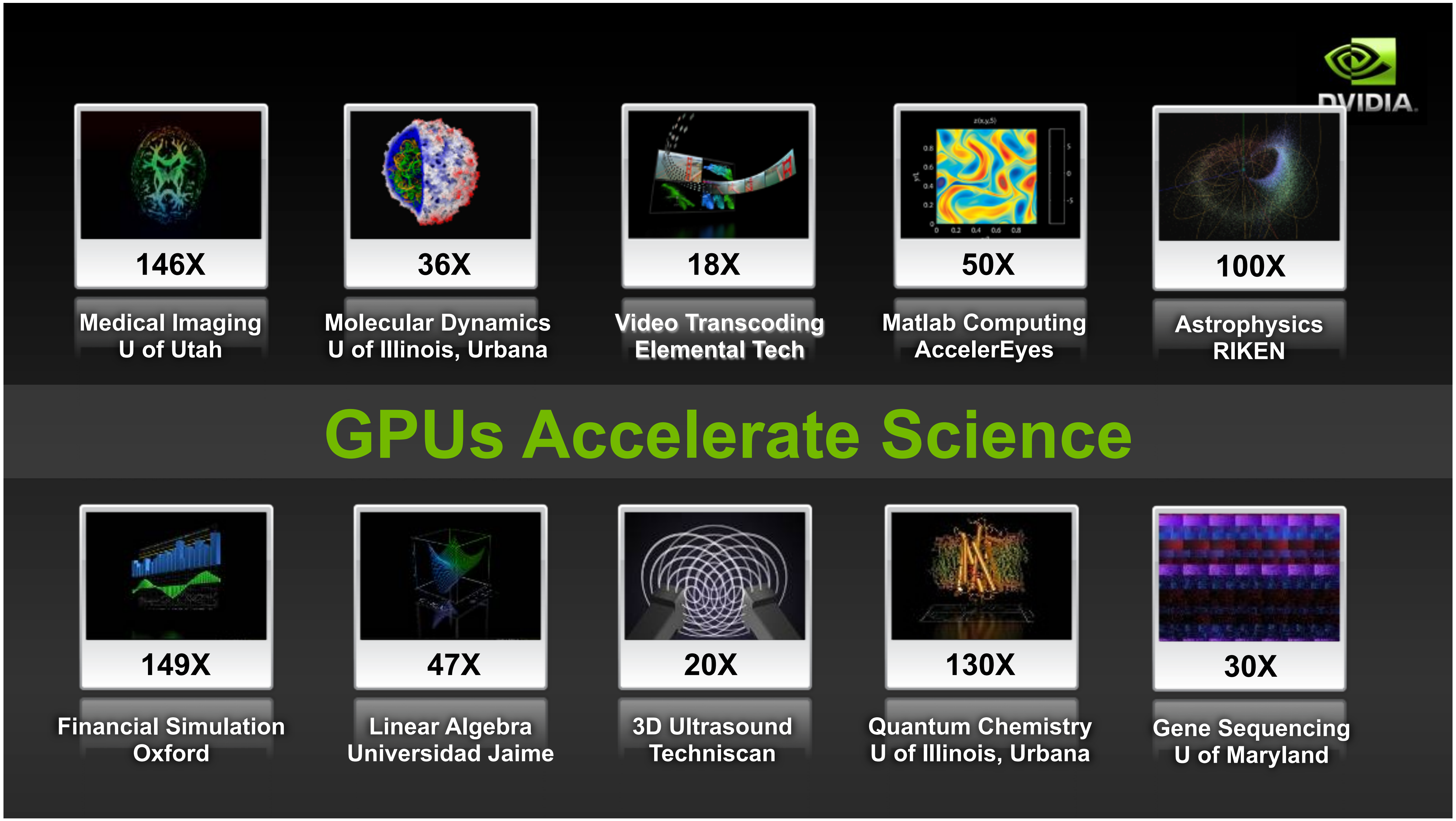
***only 4 nodes, each with 2 GPUs, 2 8-core Intel Sandy Bridge CPUs

Hokiespeed:

NVIDIA Tesla C2050 (Fermi) GPU
448 CUDA cores
3 GB global memory
515 Gflops

204 nodes, each with 2 GPUs and 2 6-core Intel Xeon E5645 CPUs

name a scientific problem. someone has probably written GPU code for it



146X	36X	18X	50X	100X
Medical Imaging U of Utah	Molecular Dynamics U of Illinois, Urbana	Video Transcoding Elemental Tech	Matlab Computing AccelerEyes	Astrophysics RIKEN

GPUs Accelerate Science

149X	47X	20X	130X	30X
Financial Simulation Oxford	Linear Algebra Universidad Jaime	3D Ultrasound Techniscan	Quantum Chemistry U of Illinois, Urbana	Gene Sequencing U of Maryland

http://hokiespeed.cs.vt.edu/events/hokiespeed-workshop/VT00_Introduction_to_GPU_Computing.pdf

Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU

Victor W Lee[†], Changkyu Kim[†], Jatin Chhugani[†], Michael Deisher[†], Daehyun Kim[†], Anthony D. Nguyen[†], Nadathur Satish[†], Mikhail Smelyanskiy[†], Srinivas Chennupaty[⋆], Per Hammarlund[⋆], Ronak Singhal[⋆] and Pradeep Dubey[†]

In the past few years there have been many studies claiming GPUs deliver substantial speedups (between 10X and 1000X) over multi-core CPUs on these kernels. To understand where such large performance difference comes from, we perform a rigorous performance analysis and find that after applying optimizations appropriate for both CPUs and GPUs the performance gap between an Nvidia GTX280 processor and the Intel Core i7 960 processor narrows to <u>only 2.5x on average</u>.

V. Lee et al. ISCA '10 Proceedings of the 37th annual international symposium on Computer architecture Pages 451–460

To be fair, though, speedups are speedups.
If your application runs 50 x faster on a GPU, that's great!

Objectives

Write a multi-GPU-enabled molecular dynamics code

Objectives

Session 1 - start simple:

1. hello world!
   - compile cuda code
   - launch simple kernel

2. simple vector addition
   - allocate cpu / gpu memory
   - cpu / gpu memory transfer

Objectives

Session 1 - start simple:

1. hello world!
   - compile cuda code
   - launch simple kernel

2. simple vector addition
   - allocate cpu / gpu memory
   - cpu / gpu memory transfer

Session 2:

3. lennard-jones forces
   - shared vs global device memory

4. building a molecular dynamics code

Objectives

Session 1 - start simple:

1. hello world!
   - compile cuda code
   - launch simple kernel

2. simple vector addition
   - allocate cpu / gpu memory
   - cpu / gpu memory transfer

Session 2:

3. lennard-jones forces
   - shared vs global device memory

4. building a molecular dynamics code

Session 3 - multiple GPUs

5. tiled DGEMM

6. multi-GPU MD

Objectives

Session 1 - start simple:

1. hello world!
   - compile cuda code
   - launch simple kernel

2. simple vector addition
   - allocate cpu / gpu memory
   - cpu / gpu memory transfer

Session 2:

3. lennard-jones forces
   - shared vs global device memory

4. building a molecular dynamics code

Session 3 - multiple GPUs

5. tiled DGEMM

6. multi-GPU MD

Session 4 - wrap up!

finish MD codes and try
accelerating your own codes

How can we use GPUs?

Three strategies:

1. open acc pragma statements

2. Use GPU versions of standard libraries (cuBLAS, cuFFT, etc.)

3. Translate C/C++/Fortran code into CUDA

How can we use GPUs?

Three strategies:

1. OpenACC directives

2. Use GPU versions of standard libraries (cuBLAS, cuFFT, etc.)

3. Translate C/C++/Fortran code into CUDA

How can we use GPUs?

Stolen from: https://developer.nvidia.com/openacc
(so don't judge my indentation)

```
#include <stdio.h>
#define N 1000000

int main(void) {
double pi = 0.0f; long i;
#pragma acc parallel loop reduction(+:pi)
for (i=0; i<N; i++) {
   double t= (double)((i+0.5)/N);
   pi +=4.0/(1.0+t*t);
}
printf("pi=%16.15f\n",pi/N);
return 0;
}
```

compiler hints, just like with OpenMP

certainly easy! I don't have much (well, any) experience with OpenACC, so this part of the tutorial ends here.

How can we use GPUs?

Three strategies:

1. OpenACC directives

2. Use GPU versions of standard libraries (cuBLAS, cuFFT, etc.)

3. Translate C/C++/Fortran code into CUDA

How can we use GPUs?

Three strategies:

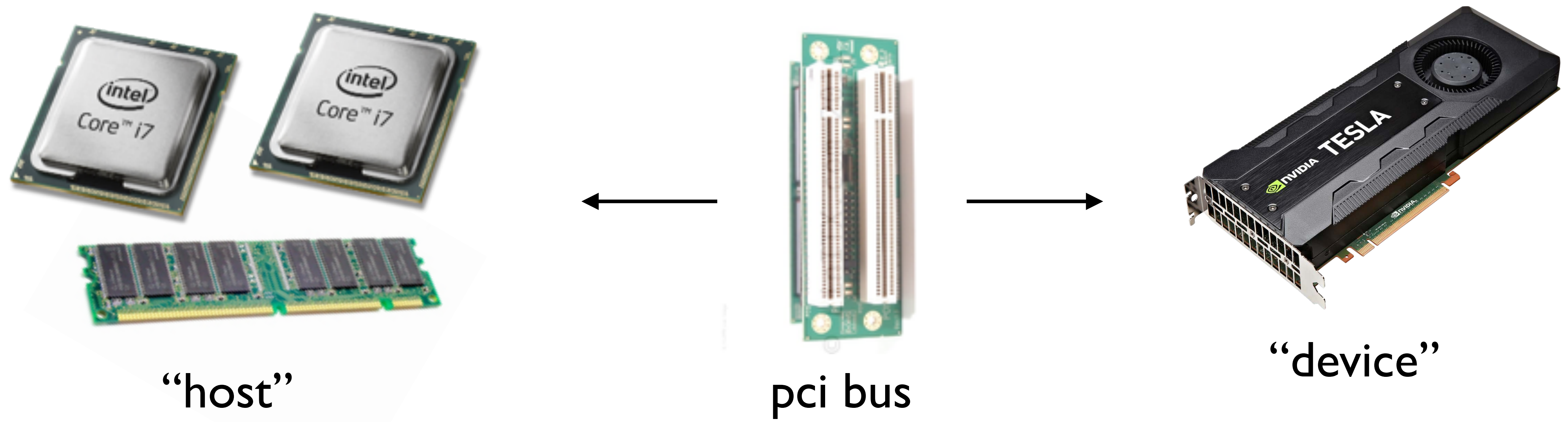
1. OpenACC directives

2. Use GPU versions of standard libraries (cuBLAS, cuFFT, etc.)

   GPU-accelerated singles and doubles coupled cluster (CCSD)

3. Translate C/C++/Fortran code into CUDA

Host vs device



rules:

1. host code (C/C++) runs on the host

2. device code (CUDA) runs on the device

3. host and device memory are separate, if you want to access data in both, you must copy it across the pci bus (expensive!)

for something like CCSD, designing an efficient algorithm comes down to masking the cost of data motion

Coupled cluster methods

The gold-standard in quantum chemistry is the coupled cluster with single and double excitations and a perturbative treatment of triple excitations: CCSD(T)

$$|\Psi_{CC}\rangle = e^{\hat{T}}|\psi_0\rangle$$

$$\hat{T} = \hat{T}_1 + \hat{T}_2 + \hat{T}_3 + \ldots + \hat{T}_N$$

$$\hat{T}_1 = \sum_{i,a} t_i^a a_a^\dagger a_i$$

$$\hat{T}_2 = \sum_{i<j,a<b} t_{ij}^{ab} a_a^\dagger a_b^\dagger a_i a_j$$

(i) formally exact if $\hat{T}$ contains all possible excitations
(ii) truncation at any order is size-extensive

Coupled cluster methods

The gold-standard in quantum chemistry is the coupled cluster with single and double excitations and a perturbative treatment of triple excitations: CCSD(T)

$$|\Psi_{CC}\rangle = e^{\hat{T}}|\psi_0\rangle$$

$$\hat{T} = \hat{T}_1 + \hat{T}_2 \qquad \bar{H} = e^{-\hat{T}}\hat{H}e^{\hat{T}}$$

$$\left.\begin{aligned} E_{CCSD} &= \langle\psi_0|\bar{H}|\Psi_0\rangle \\ 0 &= \langle\psi_S|\bar{H}|\Psi_0\rangle \\ 0 &= \langle\psi_D|\bar{H}|\Psi_0\rangle \end{aligned}\right\}$$

scales as the 6th power of system size
o^2v^4
(o=occupied, v=virtual [empty])

$E_{(T)} =$ an estimate of the energy contribution from triple excitations }

scales as the 7th power of system size
o^3v^4

7th power scaling means that doubling the systems size increases the cost of the calculation by a factor of $2^7 = 128$

One formulation of CCSD equations

$$t_{ij}^{ab} d_{ij}^{ab} = v_{ij}^{ab} + P(ia,jb) R_{ij}^{ab}$$

$$R_{ij}^{ab} += \frac{1}{2} v_{ef}^{ab} c_{ij}^{ef}$$

$$R_{ij}^{ab} += \frac{1}{2} c_{mn}^{ab} I_{ij}^{mn}$$

$$R_{ij}^{ab} -= t_{mj}^{ae} I_{ie}^{mb} + I_{ie}^{ma} t_{mj}^{eb}$$

$$R_{ij}^{ab} += (2t_{mi}^{ea} - t_{im}^{ea}) I_{ej}^{mb}$$

$$R_{ij}^{ab} += t_i^e I'^{ab}_{ej}$$

$$R_{ij}^{ab} -= t_m^a I'^{mb}_{ij}$$

$$R_{ij}^{ab} += t_{ij}^{ae} I_e^b$$

$$R_{ij}^{ab} -= t_{im}^{ab} I_j^m$$

$$I_{kl}^{ij} = v_{kl}^{ij} + v_{ef}^{ij} c_{kl}^{ef} + P(ik/jl) t_k^e v_{el}^{ij}$$

$$I_{jb}^{ia} = v_{jb}^{ia} - \frac{1}{2} v_{eb}^{im} (t_{jm}^{ea} + 2 t_j^e t_m^a) + v_{eb}^{ia} t_j^e - v_{jb}^{im} t_m^a$$

$$I'^{ab}_{ci} = v_{ci}^{ab} - v_{ci}^{am} t_m^b - v_{ci}^{mb} t_m^a$$

$$I'^{ia}_{jk} = v_{jk}^{ia} + v_{ef}^{ia} c_{jk}^{ef}$$

$$I_a^i = (2v_{ae}^{im} - v_{ea}^{im}) t_m^e$$

$$I_j^i = I'^i_j + I_e^i t_j^e$$

$$I'^i_j = (2v_{je}^{im} - v_{ej}^{im}) t_m^e + (2v_{ef}^{mi} - v_{ef}^{im}) t_{mj}^{ef}$$

$$I_b^a = (2v_{be}^{am} - v_{be}^{ma}) t_m^e - (2v_{eb}^{mn} - v_{be}^{mn}) c_{mn}^{ea}$$

$$I_{bj}^{ia} = v_{bj}^{ia} - \frac{1}{2} v_{be}^{im} (t_{mj}^{ae} + 2 t_m^a t_j^e) + v_{be}^{ia} t_j^e - v_{bj}^{im} t_m^a + \frac{1}{2} (2v_{be}^{im} - v_{eb}^{im}) t_{mj}^{ea}$$

$$t_i^a d_i^a = f_i^a + R_i^a$$

$$R_i^a += I_e^a t_i^e$$

$$R_i^a -= I'^m_i t_m^a$$

$$R_i^a += I_e^m \left(2t_{mi}^{ea} - t_{im}^{ea}\right)$$

$$R_i^a += (2v_{ei}^{ma} - v_{ei}^{am}) t_m^e$$

$$R_i^a -= v_{ei}^{mn} \left(2t_{mn}^{ea} - t_{mn}^{ae}\right)$$

$$R_i^a += v_{ef}^{ma} \left(2t_{mi}^{ef} - t_{im}^{ef}\right)$$

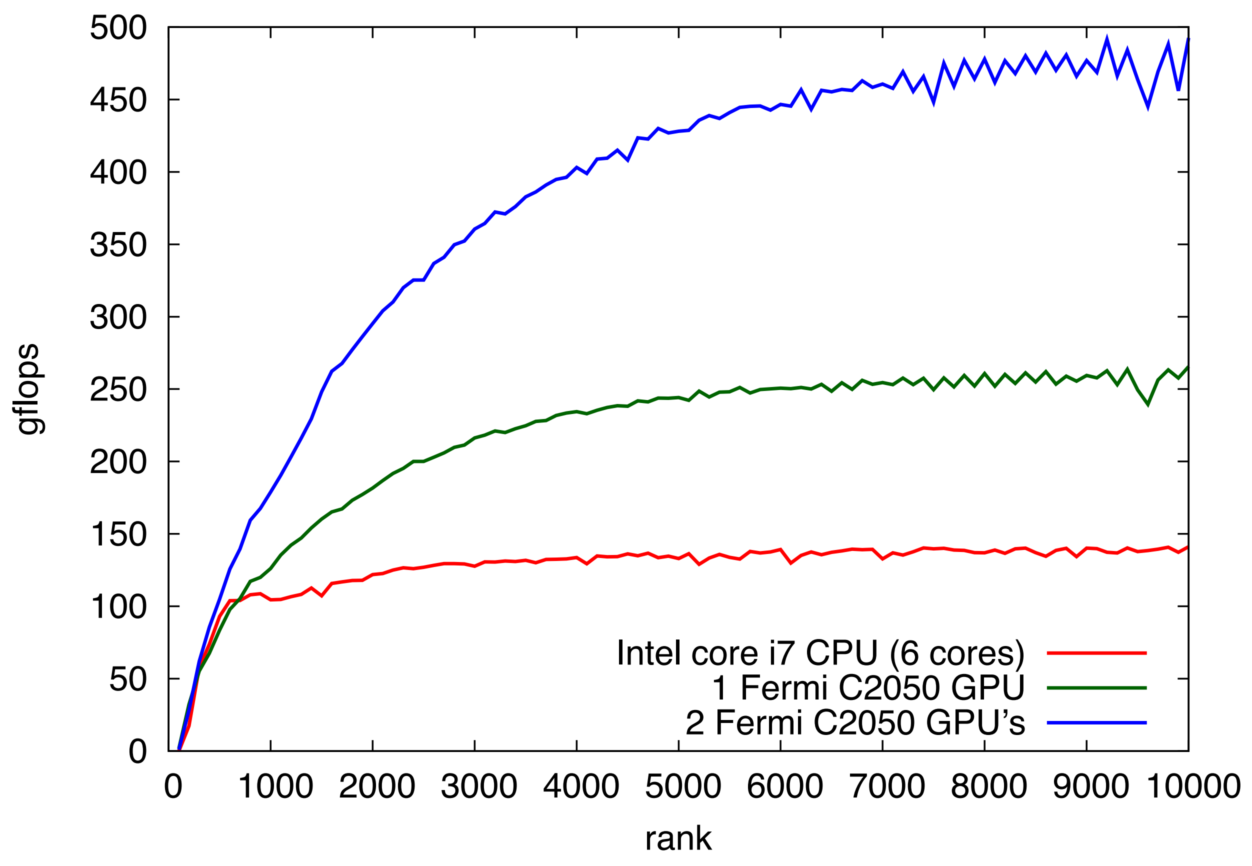
6 more contractions for singles

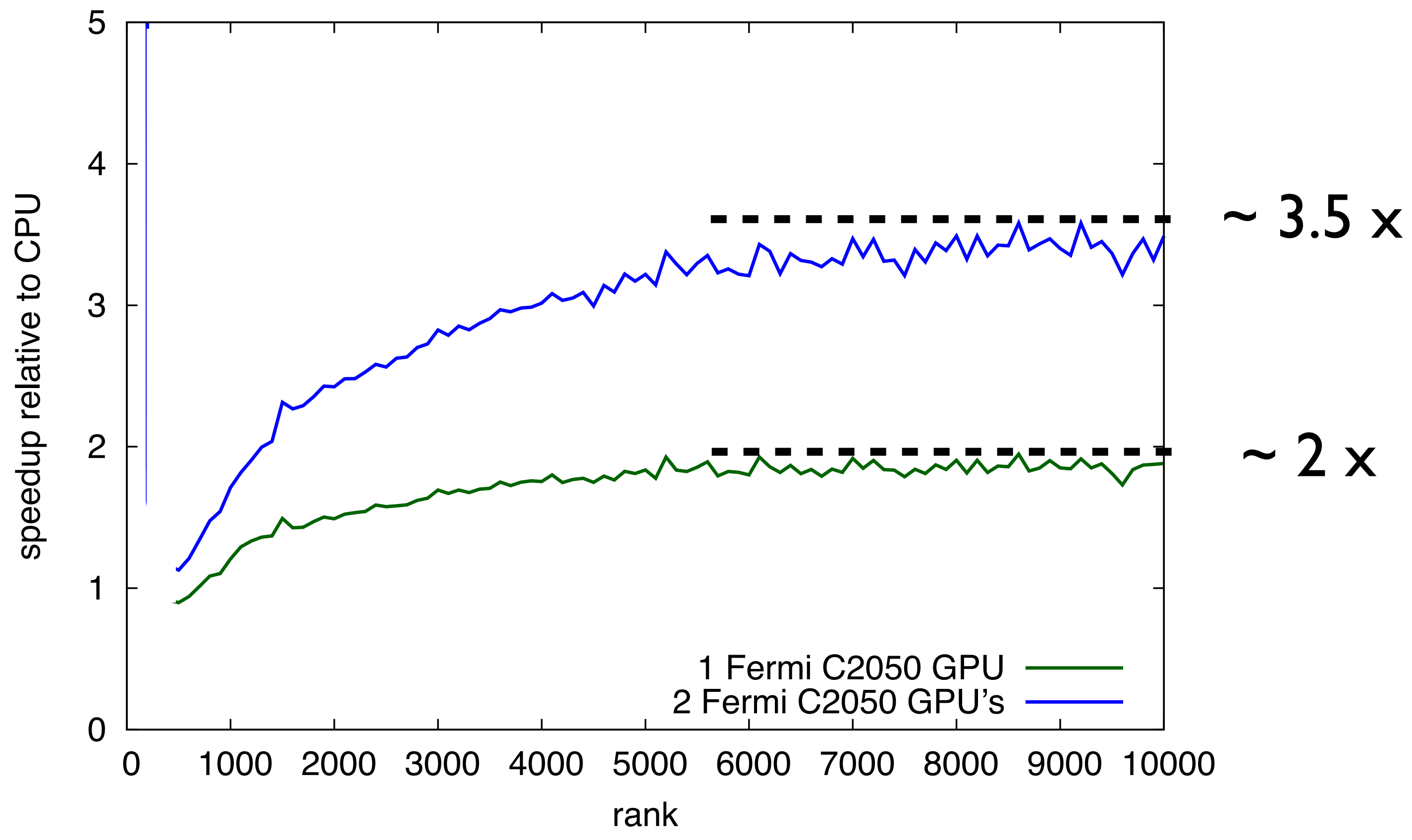
28 contractions for doubles

$$R_{ij}^{ab} += \sum_{cd} t_{ij}^{cd} \; v_{cd}^{ab}$$

CCSD boils down to a few dozen matrix-matrix multiplications.
How efficient is matrix-matrix multiplication on GPU's?

DGEMM performance

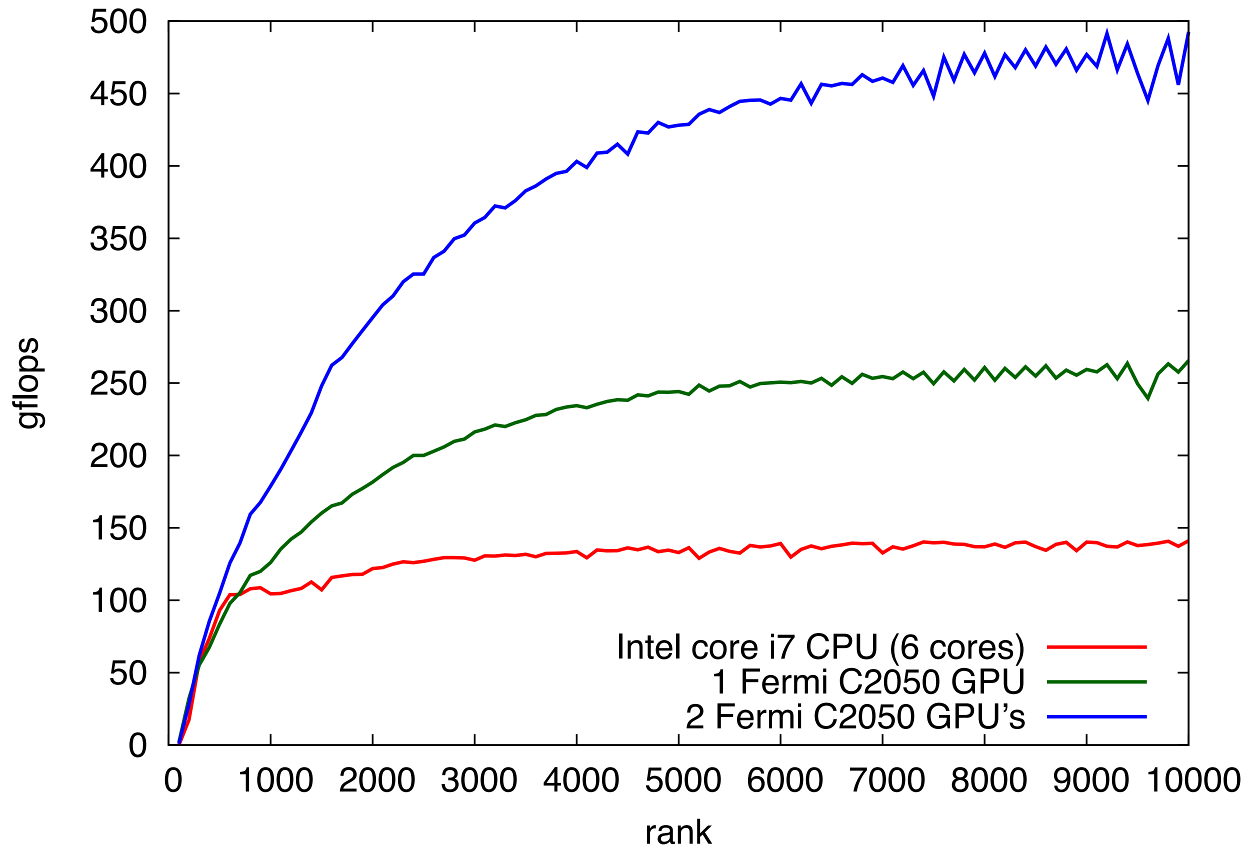


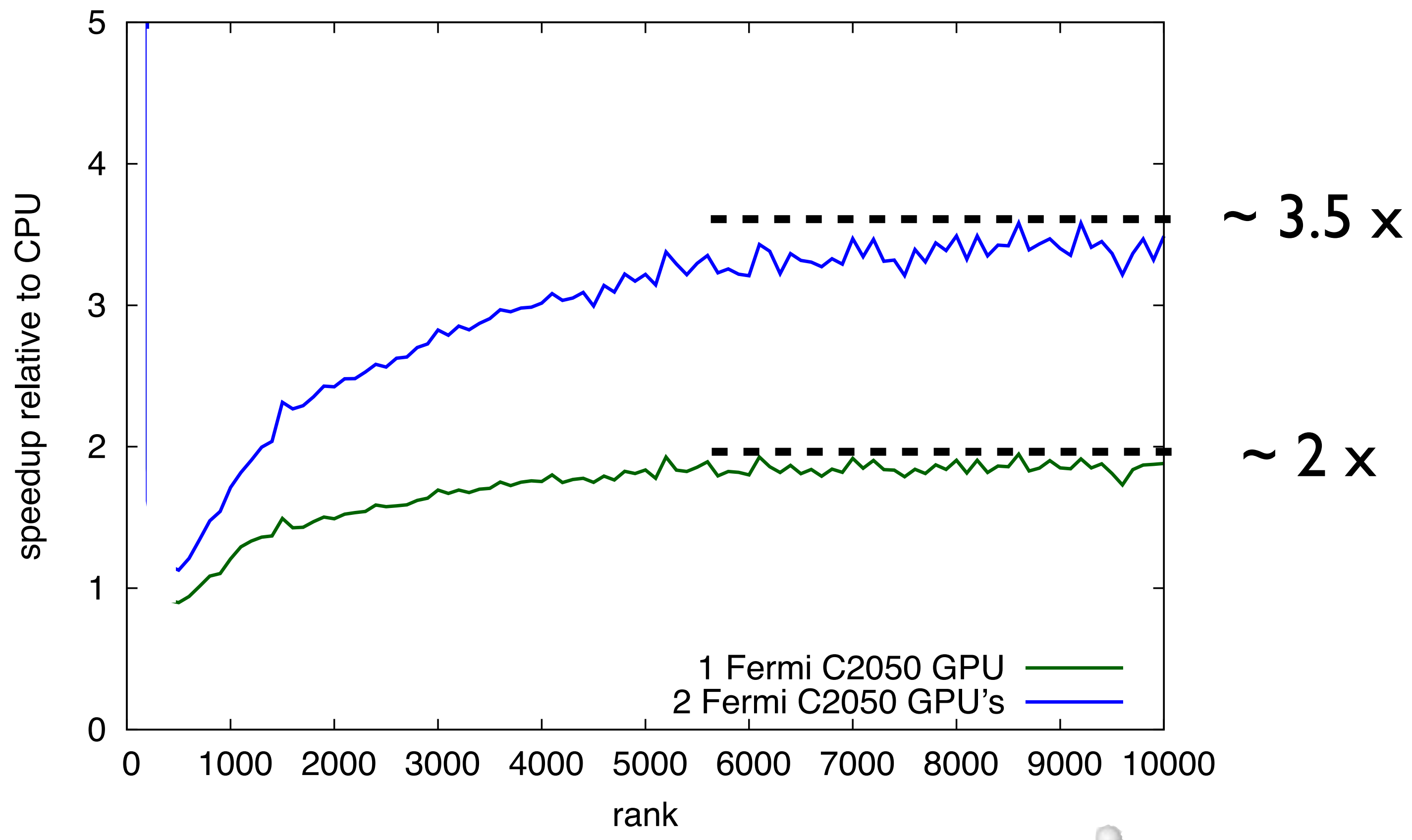


OK, let's just swap all DGEMM calls with cublasDGEMM calls in DF-CCSD

hope for the best: 2-3.5 x acceleration on this system

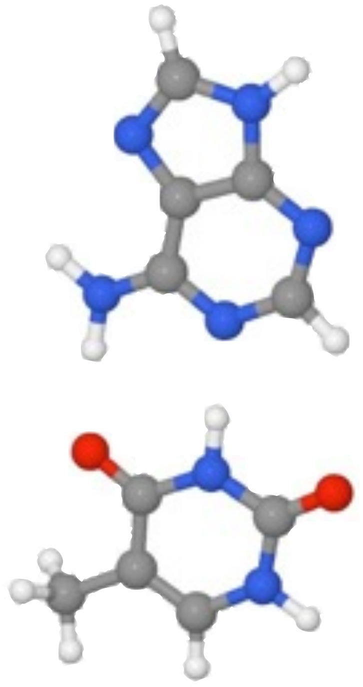
DGEMM performance





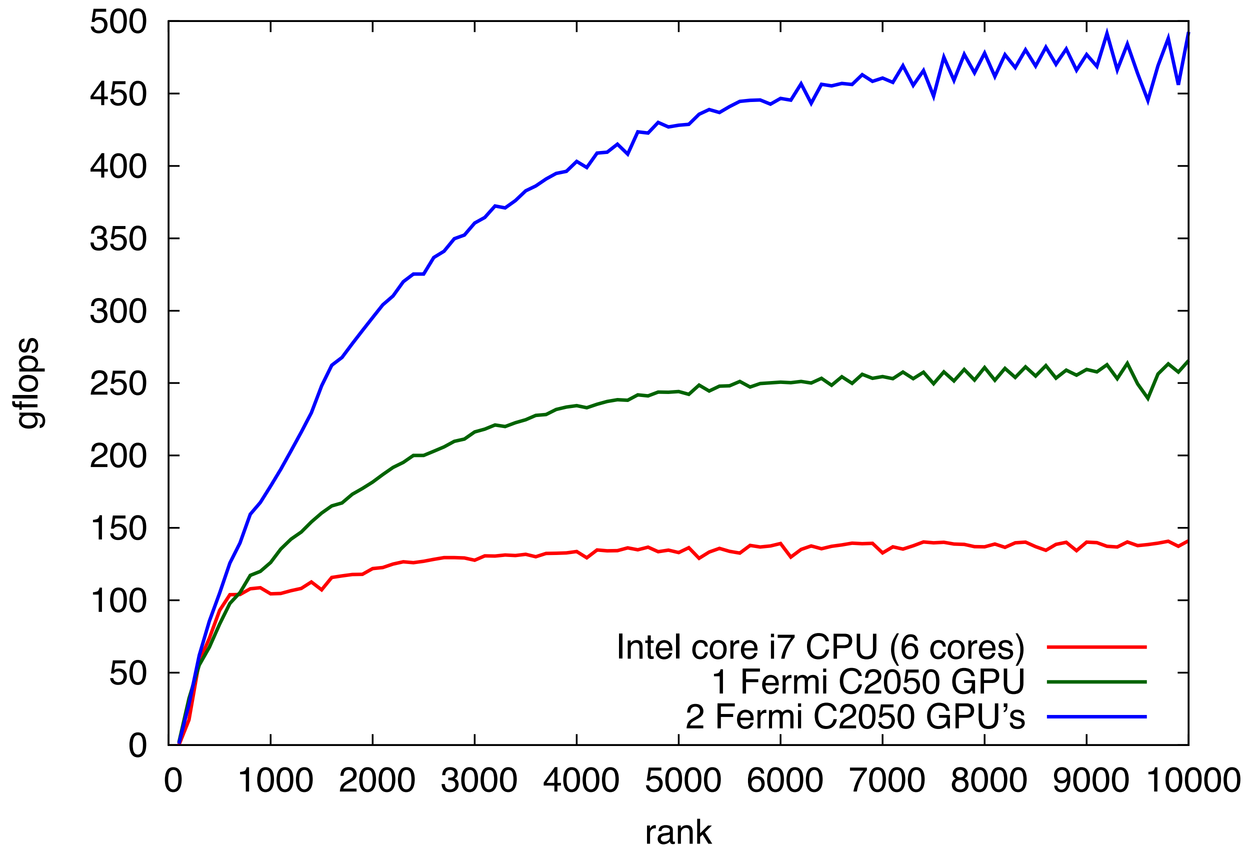
OK, let's just swap all DGEMM calls with cublasDGEMM calls in DF-CCSD

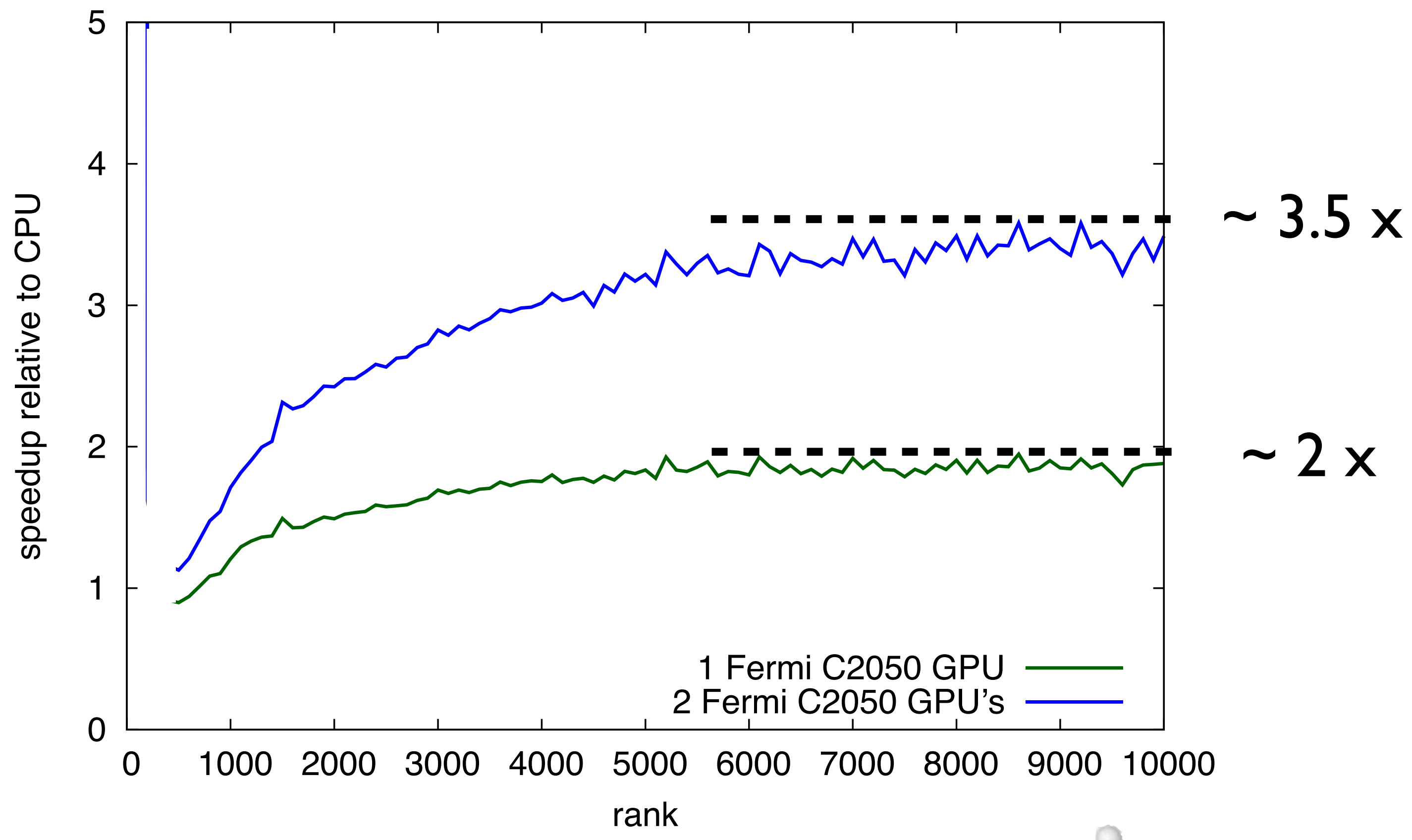
hardware	iteration time (s)	speedup
6 Intel Core i7 CPU cores	2137	
1 Fermi GPU		
2 Fermi GPU's		



adenine-thymine
aug-cc-pvdz

DGEMM performance

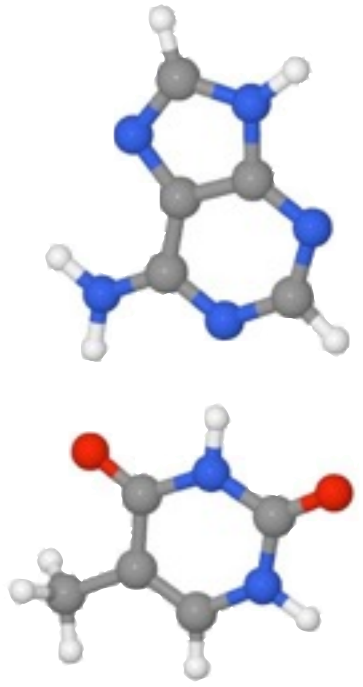




OK, let's just swap all DGEMM calls with cublasDGEMM calls in DF-CCSD

hardware	iteration time (s)	speedup
6 Intel Core i7 CPU cores	2137	
1 Fermi GPU	1817	1.18
2 Fermi GPU's	1142	1.87

well, that's disappointing.

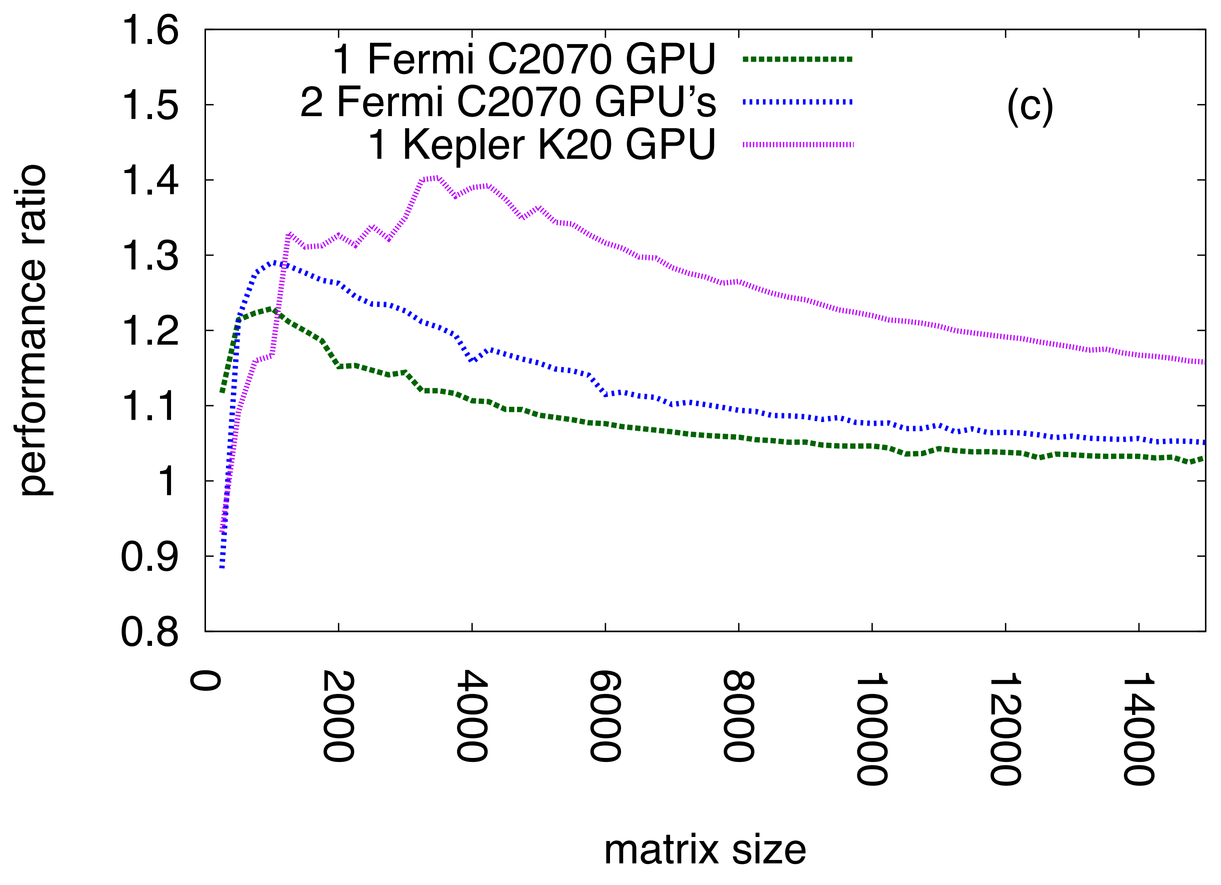


adenine-thymine
aug-cc-pvdz

What went wrong?

1. communication overhead!

   - overlapping communication and computation boosts efficiency by as much as 1.4x (more later today)



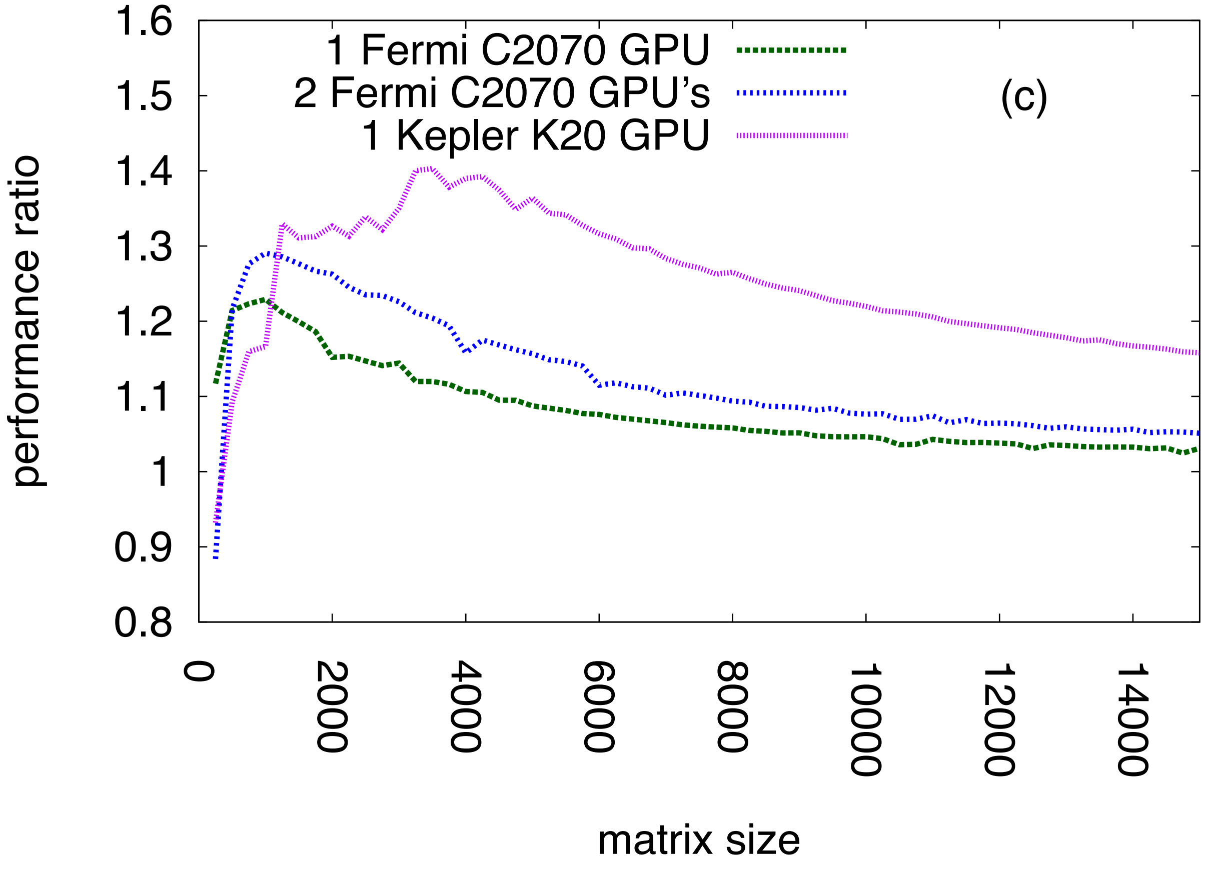
What went wrong?

I. communication overhead!

- overlapping communication and computation boosts efficiency by as much as 1.4x (more later today)

II. our CPU cores aren't doing anything!

- load balancing

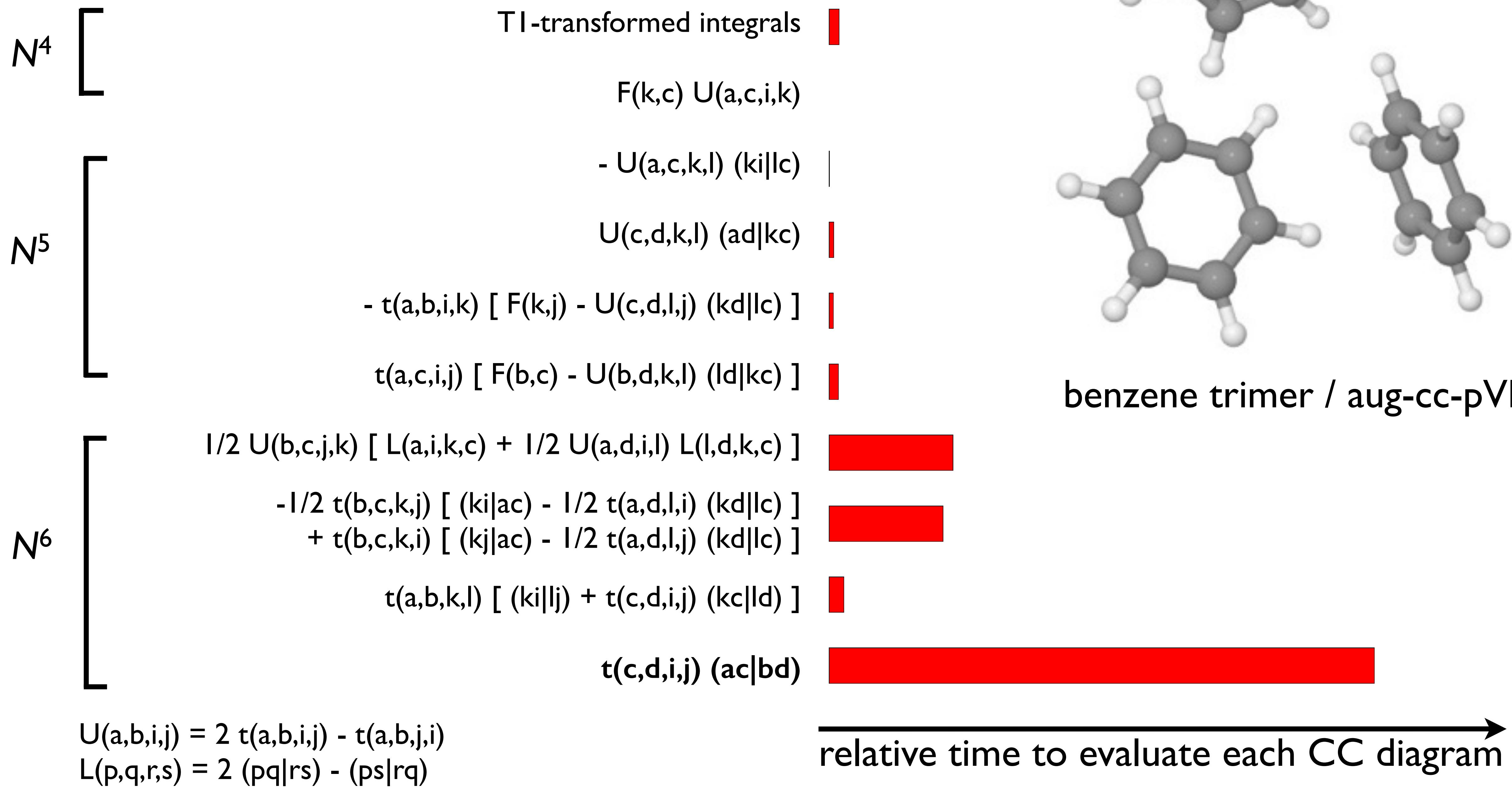


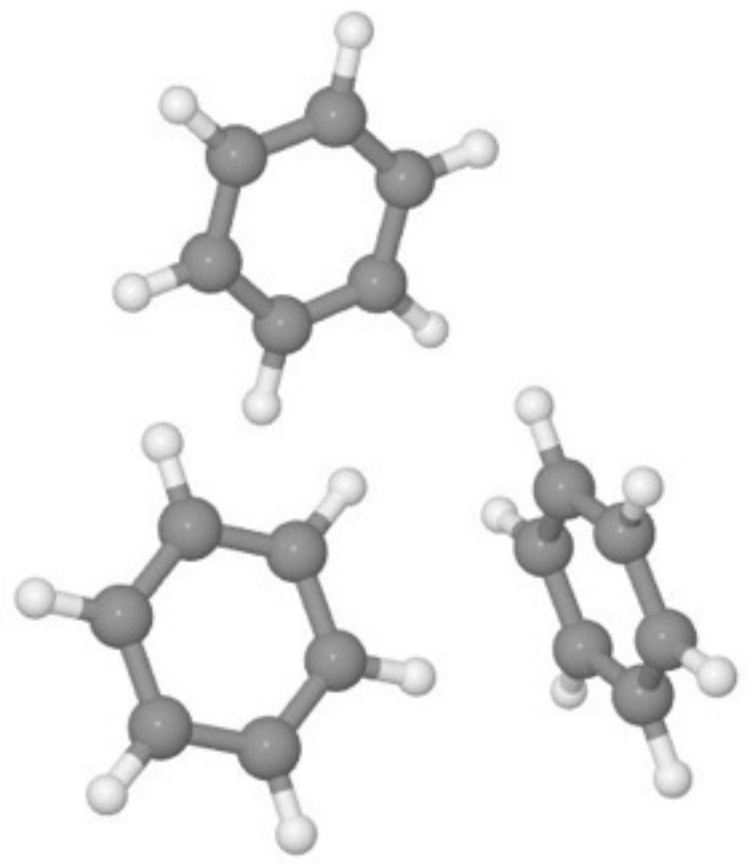
$$R_{ij}^{ab} += \frac{1}{2} v_{ef}^{ab} c_{ij}^{ef}$$
$$R_{ij}^{ab} += \frac{1}{2} c_{mn}^{ab} I_{ij}^{mn}$$
$$R_{ij}^{ab} -= t_{mj}^{ae} I_{ie}^{mb} + I_{ie}^{ma} t_{mj}^{eb}$$
$$R_{ij}^{ab} += (2t_{mi}^{ea} - t_{im}^{ea}) I_{ej}^{mb}$$
$$R_{ij}^{ab} += t_i^e I_{ej}'^{ab}$$
$$R_{ij}^{ab} -= t_m^a I_{ij}'^{mb}$$
$$R_{ij}^{ab} += t_{ij}^{ae} I_e^b$$
$$R_{ij}^{ab} -= t_{im}^{ab} I_j^m$$

$$I_{kl}^{ij} = v_{kl}^{ij} + v_{ef}^{ij} c_{kl}^{ef} + P(ik/jl) t_k^e v_{el}^{ij}$$
$$I_{jb}^{ia} = v_{jb}^{ia} - \frac{1}{2} v_{eb}^{im} (t_{jm}^{ea} + 2t_j^e t_m^a) + v_{eb}^{ia} t_j^e - v_{jb}^{im} t_m^a$$
$$I_{ci}'^{ab} = v_{ci}^{ab} - v_{ci}^{am} t_m^b - v_{ci}^{mb} t_m^a$$
$$I_{jk}'^{ia} = v_{jk}^{ia} + v_{ef}^{ia} c_{jk}^{ef}$$
$$I_a^i = (2v_{ae}^{im} - v_{ea}^{im}) t_m^e$$
$$I_j^i = I_j'^i + I_e^i t_j^e$$
$$I_j'^i = (2v_{je}^{im} - v_{ej}^{im}) t_m^e + (2v_{ef}^{mi} - v_{ef}^{im}) t_{mj}^{ef}$$
$$I_b^a = (2v_{be}^{am} - v_{be}^{ma}) t_m^e - (2v_{eb}^{mn} - v_{be}^{mn}) c_{mn}^{ea}$$
$$I_{bj}^{ia} = v_{bj}^{ia} - \frac{1}{2} v_{be}^{im} (t_{mj}^{ae} + 2t_m^a t_j^e) + v_{be}^{ia} t_j^e - v_{bj}^{im} t_m^a + \frac{1}{2} (2v_{be}^{im} - v_{eb}^{im}) t_{mj}^{ea}$$

$$t_i^a d_i^a = f_i^a + R_i^a$$
$$R_i^a += I_e^a t_i^e$$
$$R_i^a -= I_i'^m t_m^a$$
$$R_i^a += I_e^m \left(2t_{mi}^{ea} - t_{im}^{ea}\right)$$
$$R_i^a += (2v_{ei}^{ma} - v_{ei}^{am}) t_m^e$$
$$R_i^a -= v_{ei}^{mn} \left(2t_{mn}^{ea} - t_{mn}^{ae}\right)$$
$$R_i^a += v_{ef}^{ma} \left(2t_{mi}^{ef} - t_{im}^{ef}\right)$$

CCSD timings





benzene trimer / aug-cc-pVDZ

U(a,b,i,j) = 2 t(a,b,i,j) - t(a,b,j,i)
L(p,q,r,s) = 2 (pq|rs) - (ps|rq)

iteration times dominated by o^2v^4 diagram

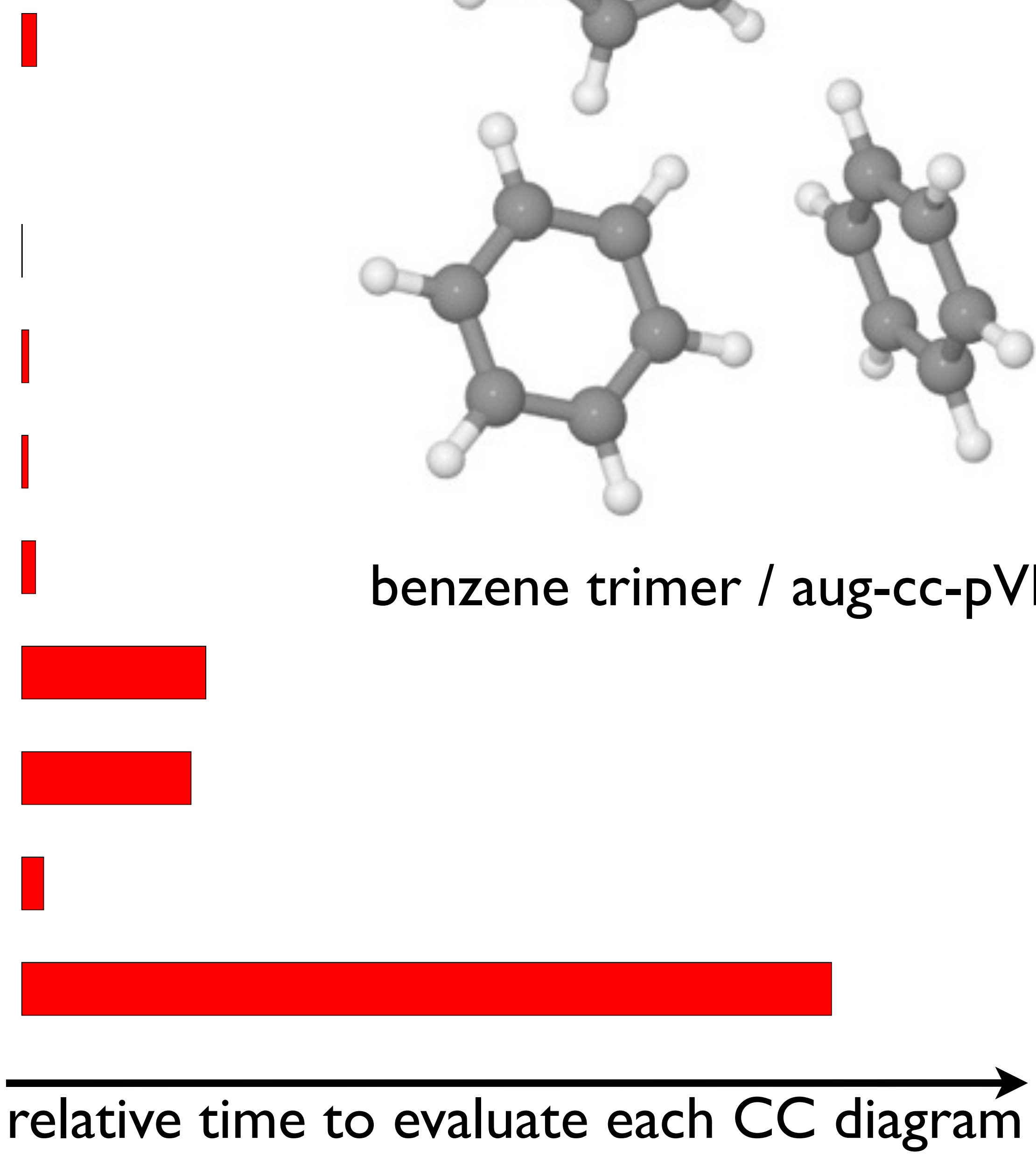
$$(ac|bd) = \sum_Q B^Q_{ac} B^Q_{bd}$$ ~$v^4 N_{aux}$ (note: we're using density fitting)

CCSD timings

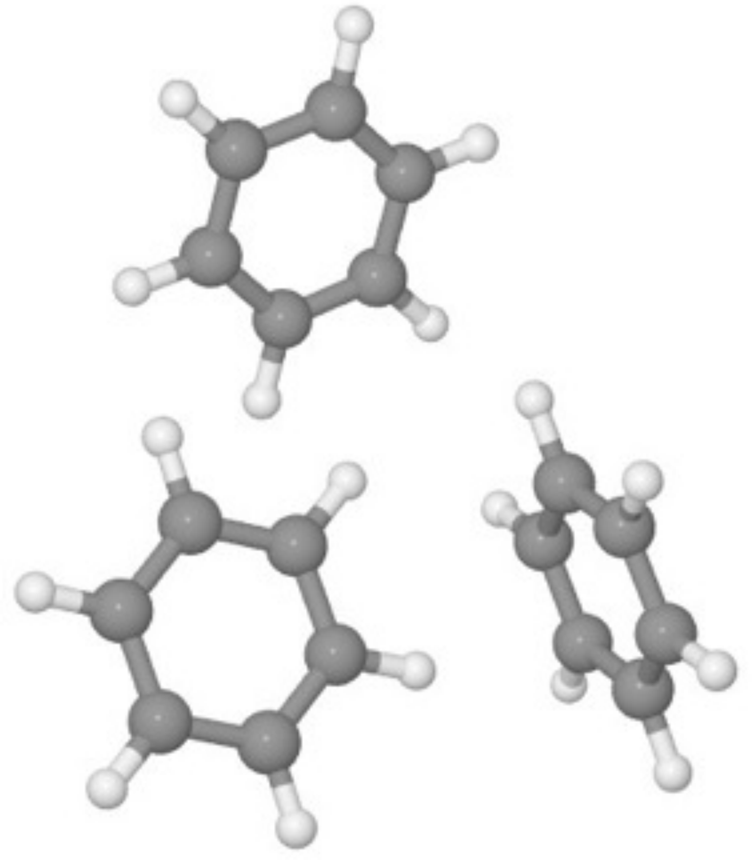
N^4
- T1-transformed integrals
- C1 F(k,c) U(a,c,i,k)

N^5
- B1 - U(a,c,k,l) (ki|lc)
- A1 U(c,d,k,l) (ad|kc)
- F2 - t(a,b,i,k) [F(k,j) - U(c,d,l,j) (kd|lc)]
- E2 t(a,c,i,j) [F(b,c) - U(b,d,k,l) (ld|kc)]

N^6
- D2 1/2 U(b,c,j,k) [L(a,i,k,c) + 1/2 U(a,d,i,l) L(l,d,k,c)]
- C2 -1/2 t(b,c,k,j) [(ki|ac) - 1/2 t(a,d,l,i) (kd|lc)]
  + t(b,c,k,i) [(kj|ac) - 1/2 t(a,d,l,j) (kd|lc)]
- B2 t(a,b,k,l) [(ki|lj) + t(c,d,i,j) (kc|ld)]
- A2 **t(c,d,i,j) (ac|bd)**



U(a,b,i,j) = 2 t(a,b,i,j) - t(a,b,j,i)
L(p,q,r,s) = 2 (pq|rs) - (ps|rq)



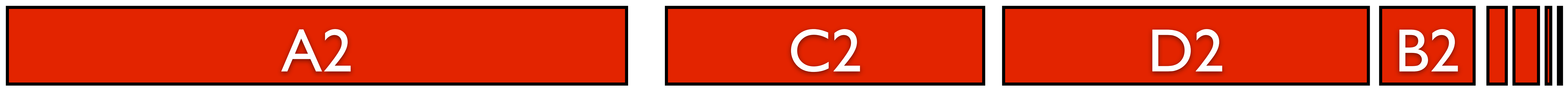
benzene trimer / aug-cc-pVDZ

iteration times dominated by o^2v^4 diagram

$$(ac|bd) = \sum_Q B^Q_{ac} B^Q_{bd}$$ ~v^4N_{aux} (note: we're using density fitting)

Implementation - a load-balancing nightmare

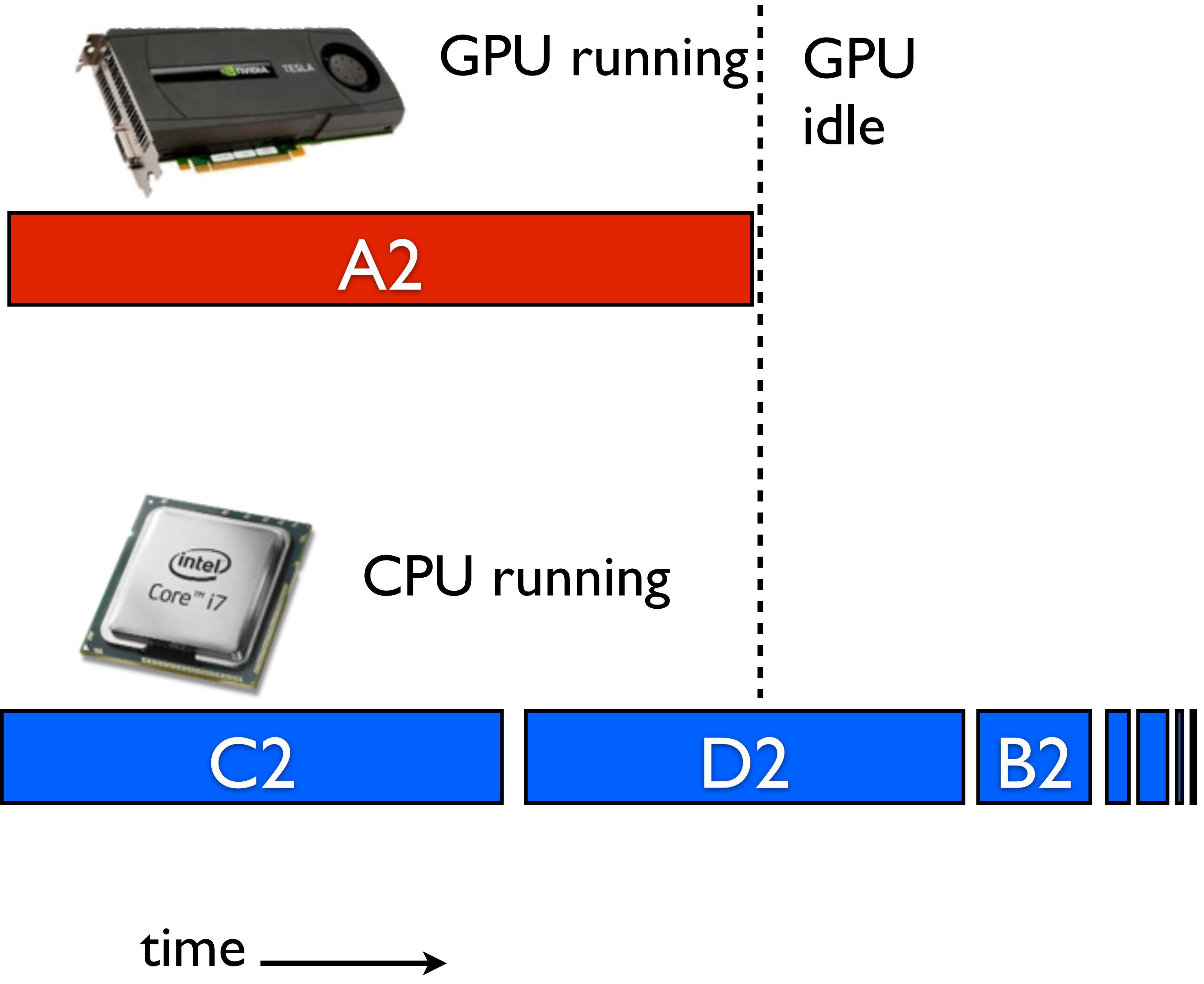
GPU running



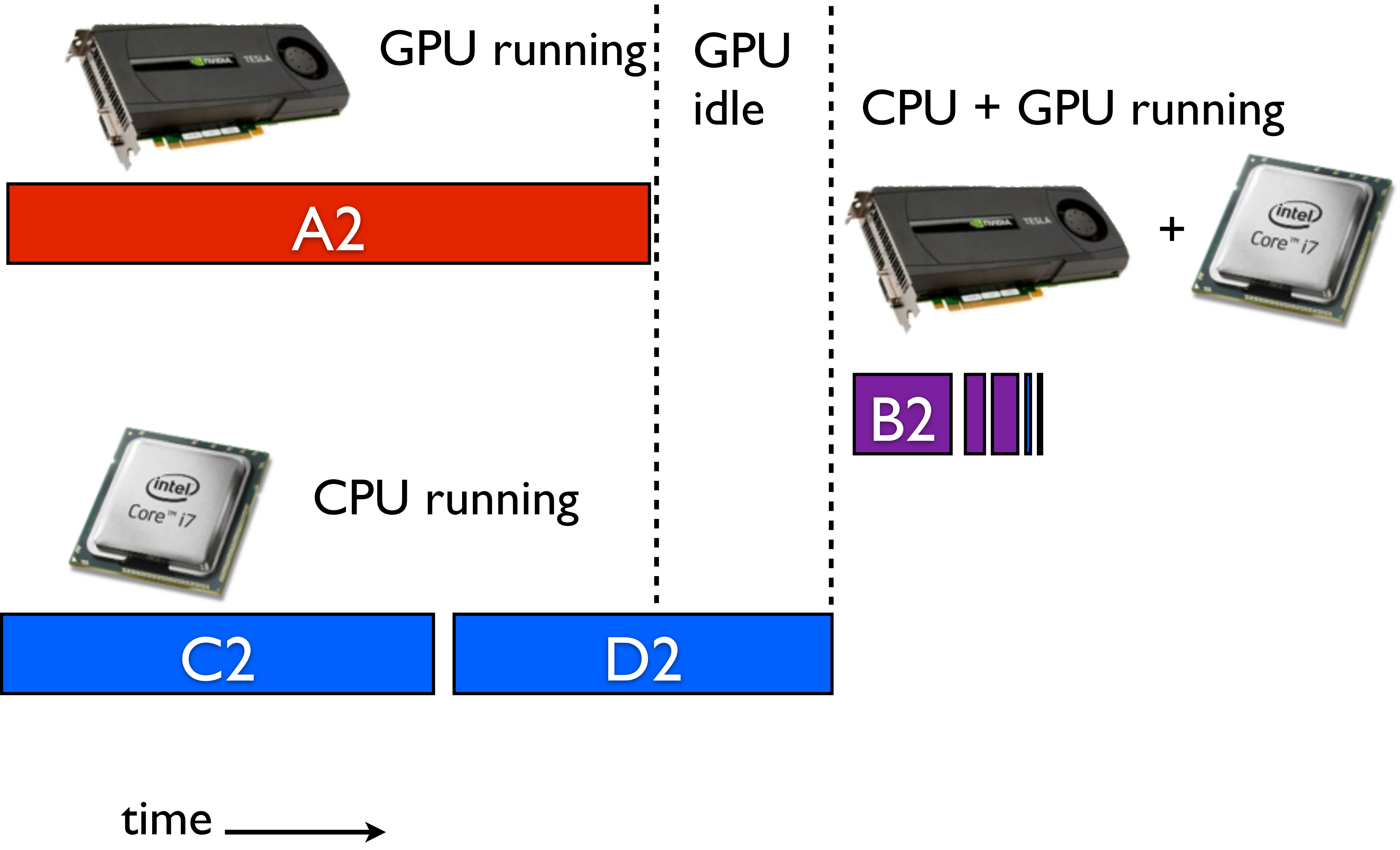
CPU idle

time $\longrightarrow$

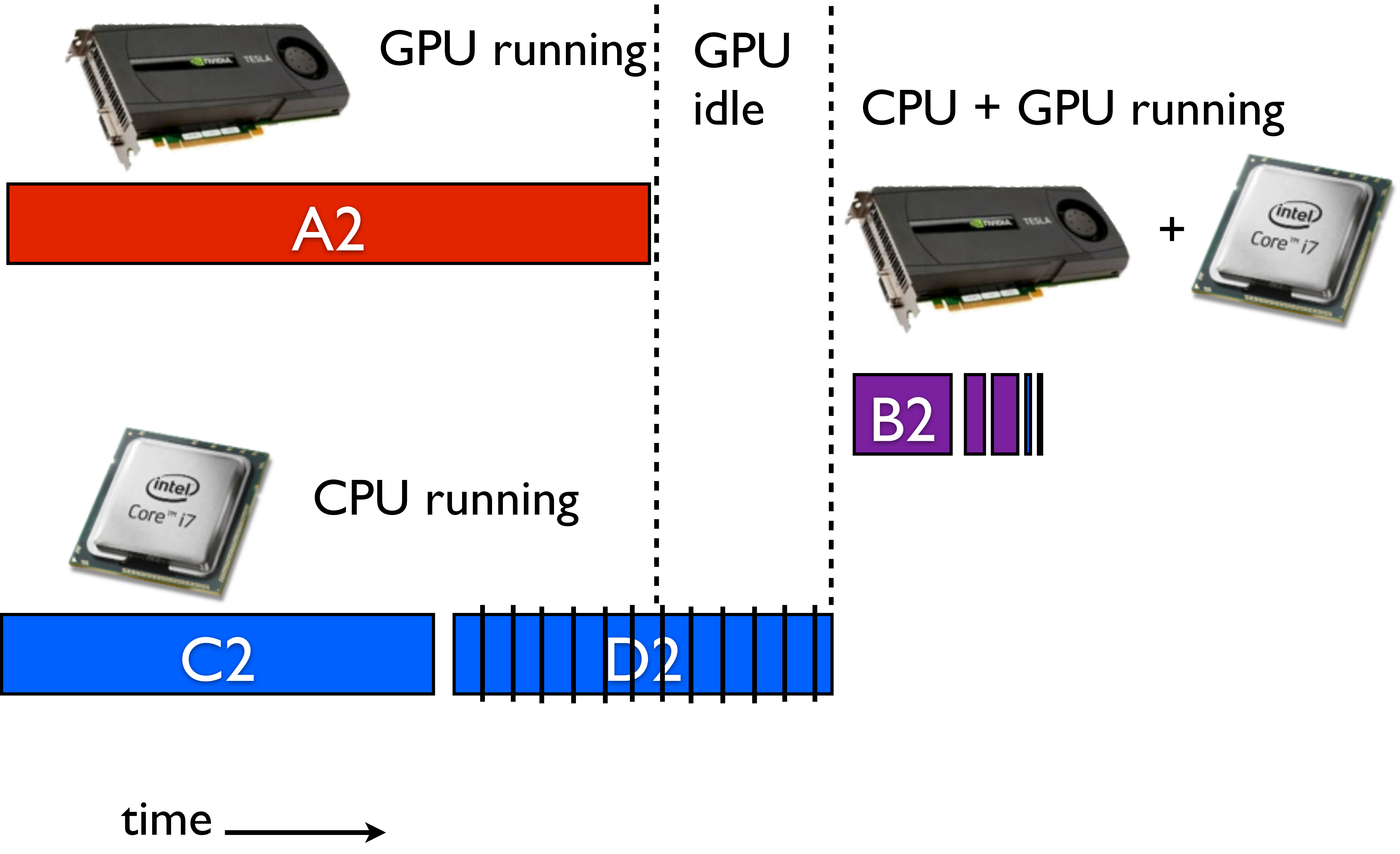
Implementation - a load-balancing nightmare



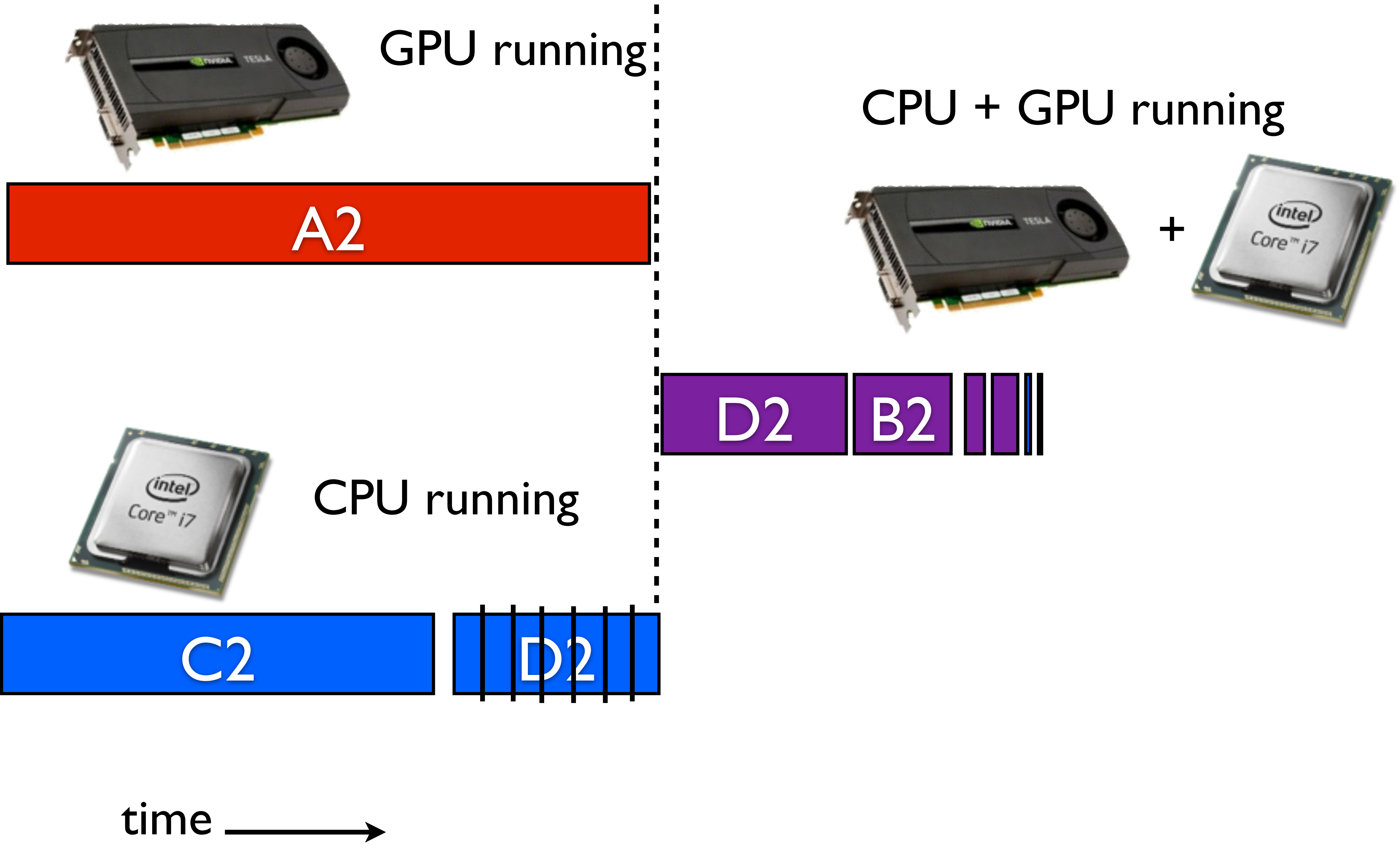
Implementation - a load-balancing nightmare

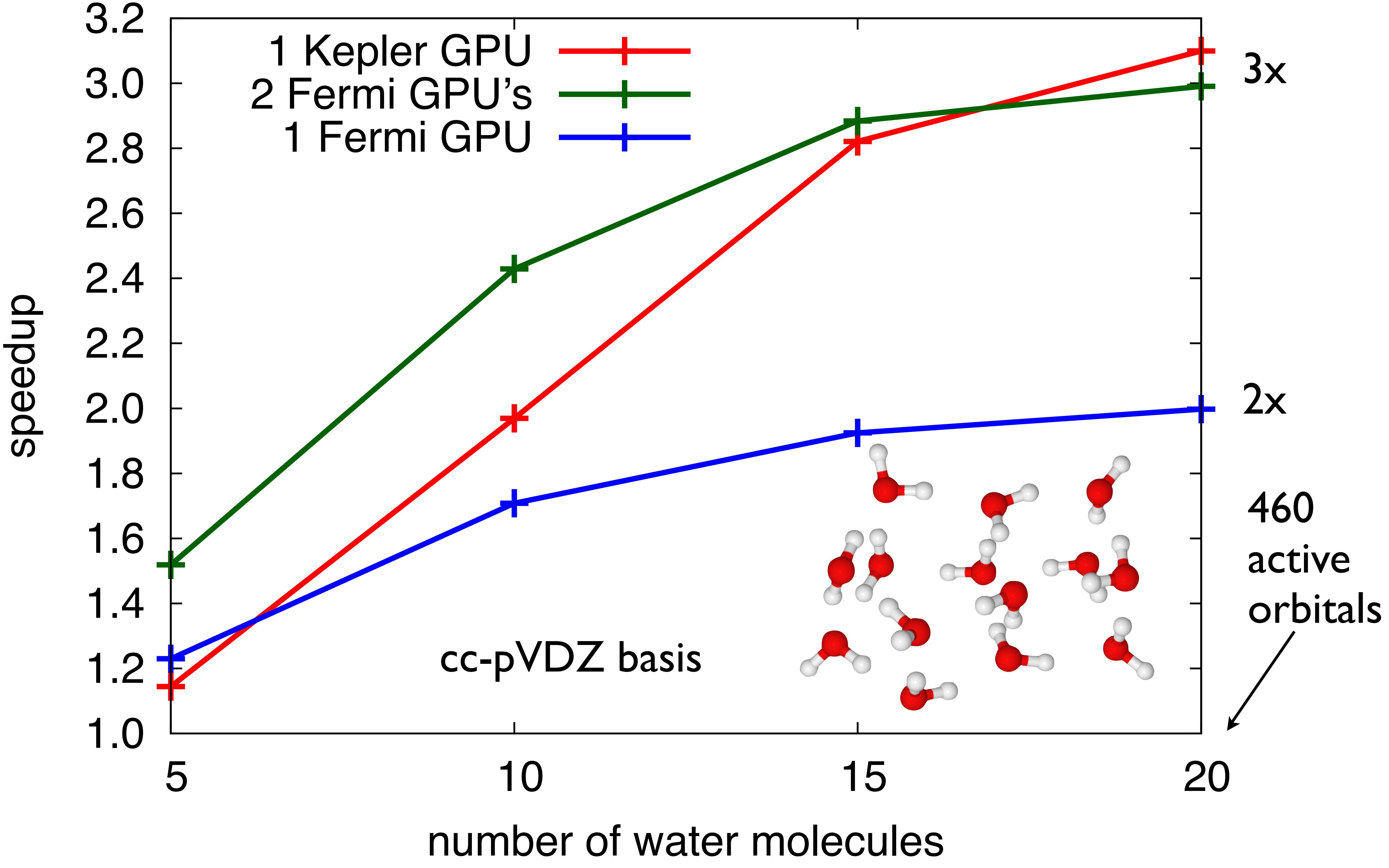


Implementation - a load-balancing nightmare



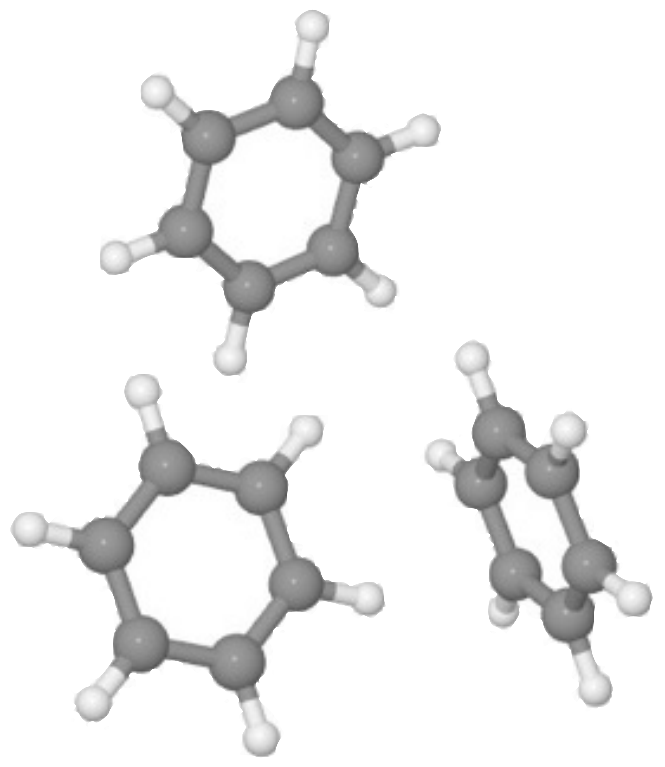
Implementation - a load-balancing nightmare



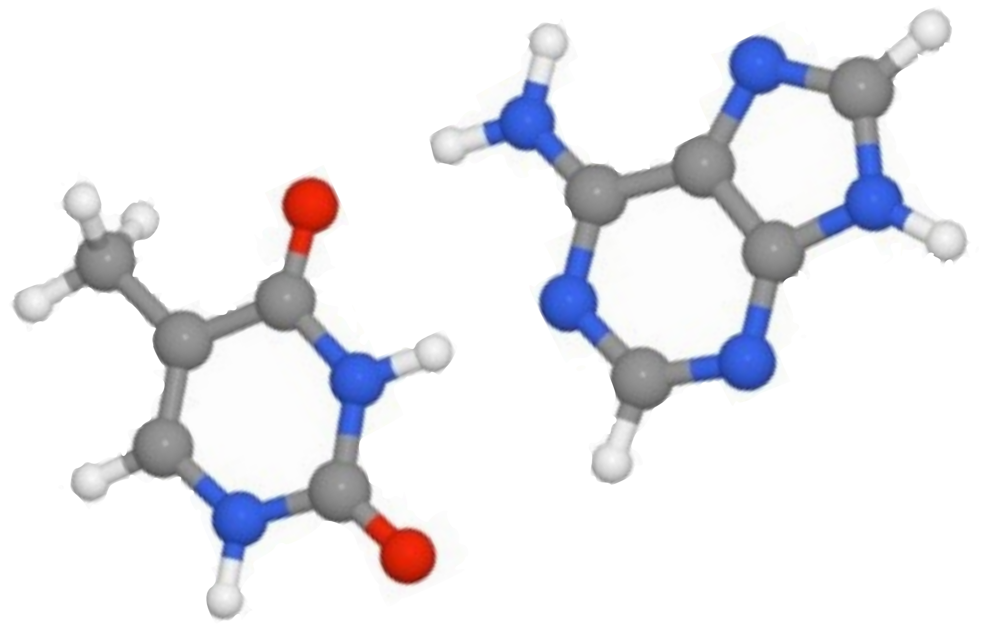

1 Kepler GPU
2 Fermi GPU's
1 Fermi GPU
speedup
number of water molecules
3x
2x
460 active orbitals
cc-pVDZ basis
3.2
3.0
2.8
2.6
2.4
2.2
2.0
1.8
1.6
1.4
1.2
1.0
5
10
15
20

average CCSD iteration times (s)

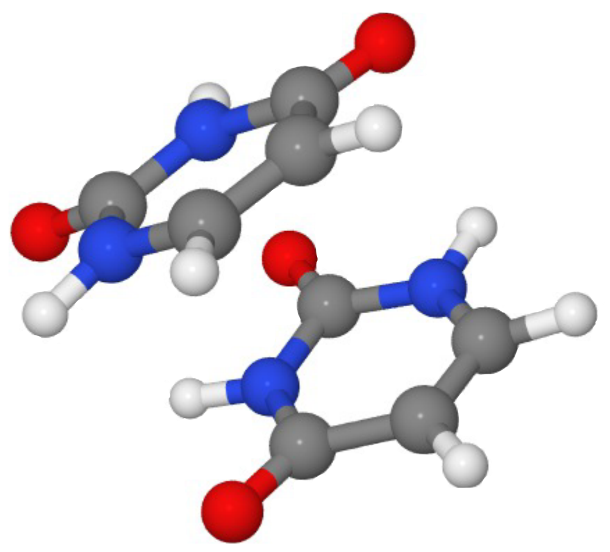
	adenine-thymine[b]			benzene trimer[c]			uracil dimer[d,e]		
	A_{ij}^{ab}	total	speedup	A_{ij}^{ab}	total	speedup	A_{ij}^{ab}	total	speedup
Core i7-3930K	1134	2156	-	1665	2719	-	8590	11924	-
1 Fermi C2070	703	1054	2.05	1082	1298	2.09	6095	6741	1.77
2 Fermi C2070	349	747	2.89	546	958	2.84	4159	4791	2.49
Kepler K20c	457	789	2.73	648	995	2.73	3826	4724	2.52



benzene trimer
aug-cc-pVDZ
558 active orbitals



adenine-thymine
aug-cc-pVDZ
517 active orbitals



uracil dimer
aug-cc-pVTZ
822 active basis functions
(using FNO's)

DePrince, Kennedy, Sumpter, and Sherrill, Mol. Phys. 112, 844-852 (2014).

How can we use GPUs?

Three strategies:

1. OpenACC directives

2. **Use GPU versions of standard libraries (cuBLAS, cuFFT, etc.)**

   at a first pass, very easy, but results can be disappointing. not surprisingly, you get better results if you think about what you're doing.

3. Translate C/C++/Fortran code into CUDA

How can we use GPUs?

Three strategies:

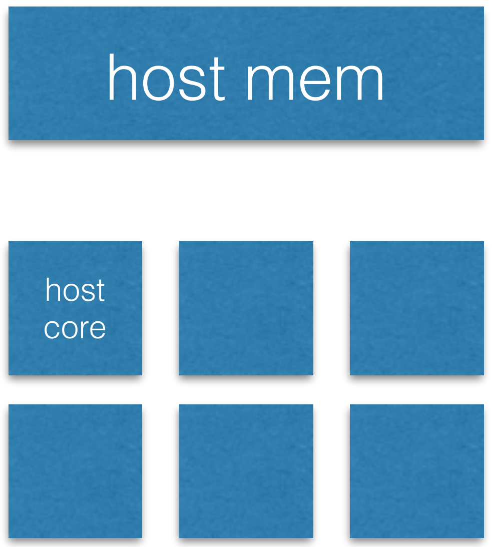
1. open acc pragma statements

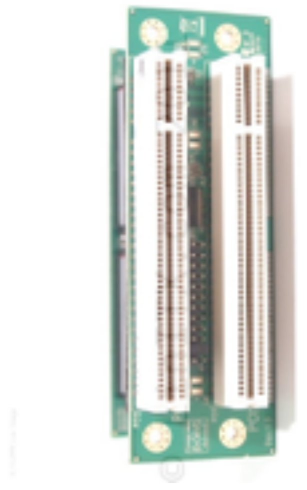
2. Use GPU versions of standard libraries (cuBLAS, cuFFT, etc.)

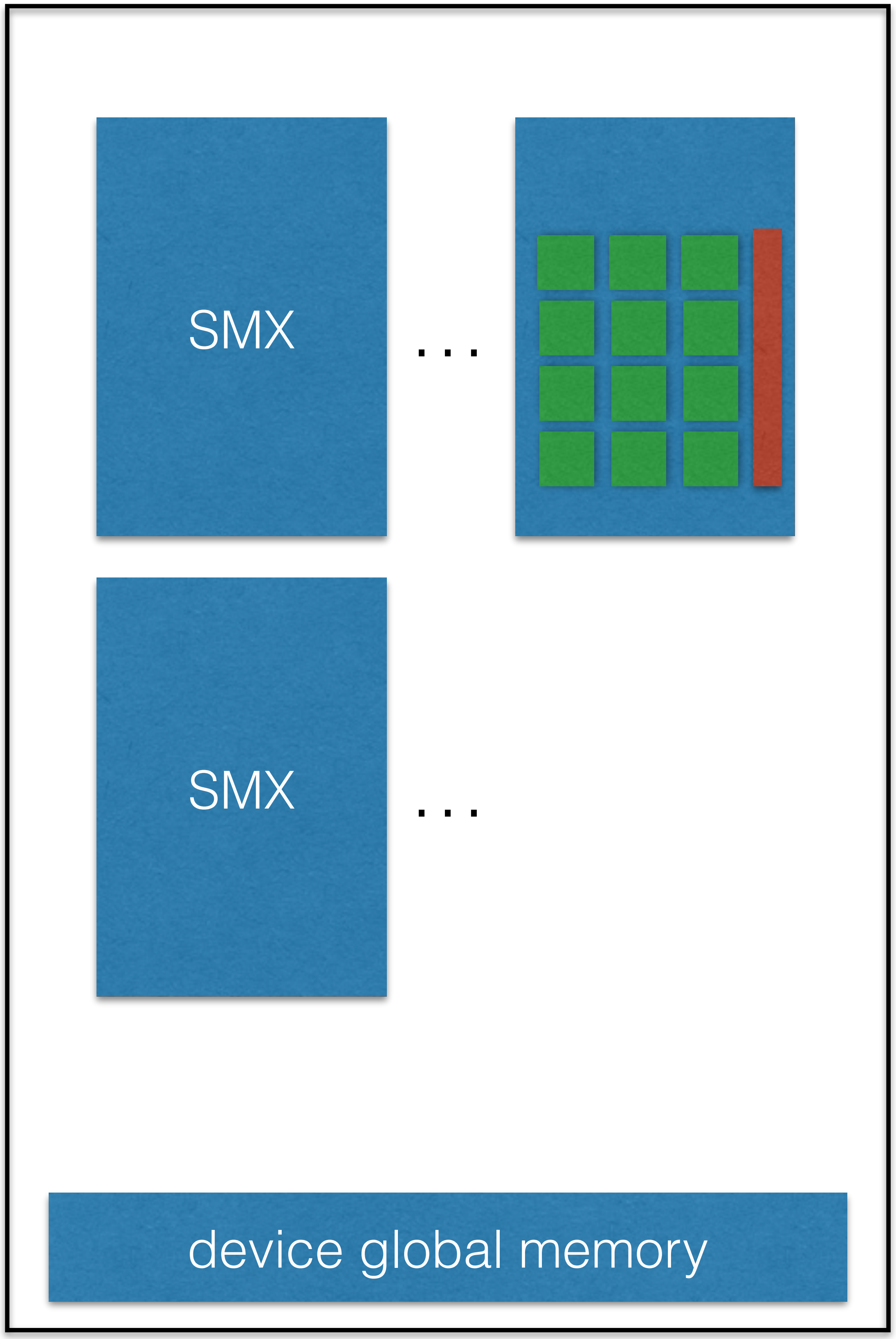
3. Translate C/C++/Fortran code into CUDA

   THIS is where the real fun is!

Program flow







```
int main() {
      host code (run on CPU)
      transfer data to device
      launch kernel on device
      device code (run on GPU)
      transfer data back to host
      return 0;
}
```

hello world!

an example code can be found in S212/gpu/hello/main.cu

you can compile this code with standard compilers or with nvcc

```
int main (int argc, char* argv[]) {

    printf(“hello from the cpu!\n”);
    return 0;

}
```

hello world!

an example code can be found in S212/gpu/hello/main.cu

```
#include<cuda.h>
#include<cuda_runtime.h>

__global__ void hello() {
}

int main (int argc, char* argv[]) {

    printf(“hello from the cpu!\n”);

    hello<<<1,1>>>();

    return 0;

}
```

hello world!

an example code can be found in S212/gpu/hello/main.cu

```
#include<cuda.h>
#include<cuda_runtime.h>

__global__ void hello() {
}

int main (int argc, char* argv[]) {

    printf("hello from the cpu!\n");

    hello<<<1,1>>>();

    return 0;

}
```

`__global__`
code run by device,
called by host

hello world!

an example code can be found in S212/gpu/hello/main.cu

```
#include<cuda.h>
#include<cuda_runtime.h>

__global__ void hello() {
}

int main (int argc, char* argv[]) {

    printf(“hello from the cpu!\n”);

    hello<<<1,1>>>();

    return 0;

}
```

brackets mean host calling device code

the 1,1 determines the number of threads launched

hello world!

an example code can be found in S212/gpu/hello/main.cu

```
#include<cuda.h>
#include<cuda_runtime.h>

__global__ void hello() {
}

int main (int argc, char* argv[]) {

    printf(“hello from the cpu!\n”);

    hello<<<1,1>>>();

    return 0;

}
```

since we have cuda code now, the file must have a .cu extension and needs to be compiled with nvcc

hello world!

an example code can be found in S212/gpu/hello/main.cu

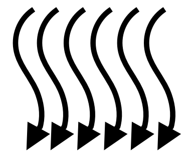
```
#include<cuda.h>
#include<cuda_runtime.h>

__global__ void hello() {
    printf("thread %5i from block %5i says, \"hello, world!\”\n”,
           threadIdx.x,blockIdx.x);

}

int main (int argc, char* argv[]) {

    printf(“hello from the cpu!\n”);

    hello<<<1,1>>>();

    return 0;

}
```

what in the world are threads and blocks??

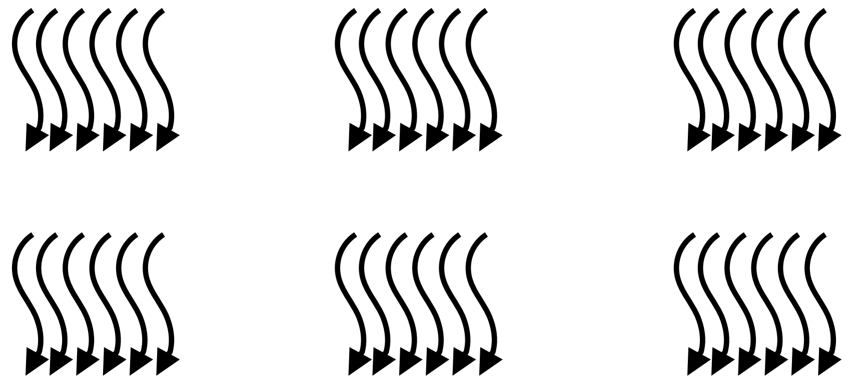
Threads, blocks, and grids

A **thread** is a sequence of instructions

On a GPU, threads are organized into **blocks**



and blocks are organized on a **grid**



Understanding this structure is important for (at least) two reasons:

1. indexing threads (threadIdx.x, blockIdx.x)
2. using shared memory within blocks

```
hello<<<1,1>>>();
```

launches the kernel with 1 block consisting of 1 thread

hello world!

an example code can be found in S212/gpu/hello/main.cu

```
__global__ void hello() {
    printf("thread %5i from block %5i says, \"hello, world!\”\n”,
           threadIdx.x,blockIdx.x);

}
int main (int argc, char* argv[]) {

    printf(“hello from the cpu!\n”);

    hello<<<1,2>>>();

    return 0;

}
```

result:
```
thread     0 from block     0 says, "hello, world!"
thread     1 from block     0 says, "hello, world!"
```

the point: each thread executes exactly the same kernel.

We can use different threads to act on different data using their `threadIdx.x`, `blockIdx.x`, etc. identifiers

now, you try.

an example code can be found in S212/gpu/hello/main.cu

(you should write your own, though!)

```
ssh hpcXXX@hokiespeed1.arc.vt.edu

/home/TRAINING/SICM2/interactive_hpcXX.sh

module load cuda

nvcc  -O2 -arch sm_20 -Xcompiler -fopenmp main.cu -o hello.x
```

hello world!

what happens if you try to execute your kernel with 1 block and 10000 threads?

```
#include<cuda.h>
#include<cuda_runtime.h>

__global__ void hello() {
    printf("thread %5i from block %5i says, \"hello, world!\”\n”,
           threadIdx.x,blockIdx.x);

}

int main (int argc, char* argv[]) {

    printf(“hello from the cpu!\n”);

    hello<<<1,10000>>>();

    return 0;

}
```

what went wrong? well, as this code is written, we have no way to know!

hello world!

```
#include<cuda.h>
#include<cuda_runtime.h>

__global__ void hello() {
    printf("thread %5i from block %5i says, \"hello, world!\”\n”,
           threadIdx.x,blockIdx.x);

}

int main (int argc, char* argv[]) {

    printf(“hello from the cpu!\n”);

    hello<<<1,10000>>>();

    // check for errors
    cudaError_t error = cudaGetLastError();
    if (error!=cudaSuccess) {
       printf("\n");
       printf("    error: %s\n", cudaGetErrorString(error) );
       printf("\n");
       exit(EXIT_FAILURE);
    }

    return 0;

}
```

Invalid configuration??

```
error: invalid configuration argument
```

as it turns out, there is a limit to the number of threads that can be executed per block and the number of blocks (in each dimension…)

```
struct cudaDeviceProp cudaProp;
int gpu_id;
cudaGetDevice(&gpu_id);
cudaGetDeviceProperties( &cudaProp,gpu_id );


printf("        maxThreadsPerBlock:   %20d\n",cudaProp.maxThreadsPerBlock);
```

will tell you the maximum number of threads allowed per block
(1024 on Fermi and Kepler).

the maximum number of blocks (per dimension) is 65535

other useful info can be found in the device properties (global / shared memory, etc.)

vector addition

the code relevant to this section can be found in

S212/gpu/vecadd/main.cu

also, you can look at the tutorial here:

http://www.chem.fsu.edu/~deprince/programming_projects/gpu_vecadd/

vector addition: **C = A + B**

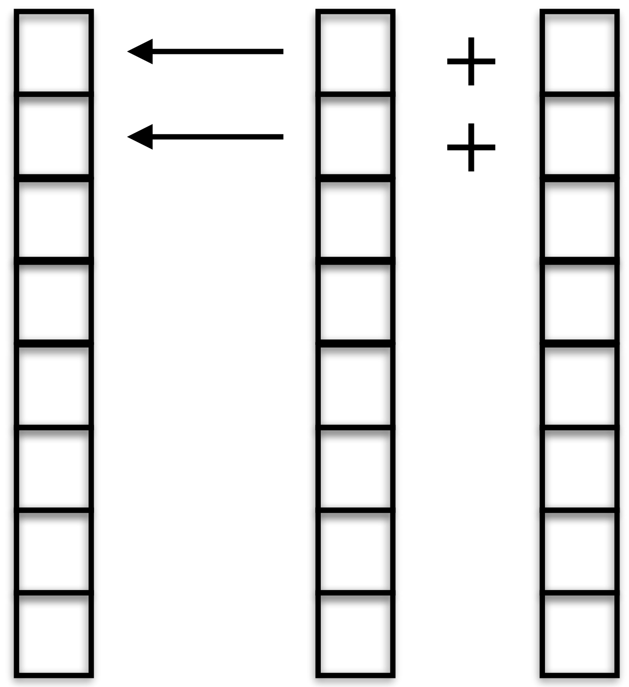
cpu code:

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < n; i++) {
    c[i] = a[i] + b[i];
}
```

vector addition

this is a very natural problem for the GPU - each thread can handle a different index in the sum!

C = A + B



gpu code:

first, we need to allocate memory for a, b, and c on the device:

```
// pointers to gpu memory
double * aGPU;
double * bGPU;
double * cGPU;

// allocate memory on gpu
cudaMalloc((void**)&aGPU,n*sizeof(double));
cudaMalloc((void**)&bGPU,n*sizeof(double));
cudaMalloc((void**)&cGPU,n*sizeof(double));
```

vector addition

gpu code:

now, we need to copy a and b to the device

```
// copy data from cpu to gpu memory
cudaMemcpy(aGPU,a,n*sizeof(double),cudaMemcpyHostToDevice);
cudaMemcpy(bGPU,b,n*sizeof(double),cudaMemcpyHostToDevice);
```

target   source   size   direction (host->device)

vector addition

gpu code:

now, let's launch a kernel that evaluates C = A + B

```
vecadd_gpu_bythread<<<1,n>>>(aGPU,bGPU,cGPU);
```

↖ launches 1 thread block with n threads

the corresponding CUDA code (outside main, of course)

```
// C = A + B, evaluated on a single block with n threads
__global__ void vecadd_gpu_bythread(double * a, double * b, double * c) {

    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

↖ the same code is executed by every thread. each thread acts on its own piece of memory, according to its `threadIdx.x`

vector addition

gpu code:

alternatively, we could parallelize over blocks:

```
vecadd_gpu_byblock<<<n,1>>>(aGPU,bGPU,cGPU);
```

launches n thread blocks, each with 1 thread

the corresponding CUDA code (outside main, of course)

```
// C = A + B, evaluated on a n blocks, each with a single thread
__global__ void vecadd_gpu_byblock(double * a, double * b, double * c) {

    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

the same code is executed by every thread. each thread acts on its own piece of memory, according to its `blockIdx.x`

vector addition

these strategies will fail for large n. there is a limit to the number of threads that can be executed per block and the number of blocks (in each dimension…)

```
struct cudaDeviceProp cudaProp;
int gpu_id;
cudaGetDevice(&gpu_id);
cudaGetDeviceProperties( &cudaProp,gpu_id );
```

```
printf("        maxThreadsPerBlock:   %20d\n",cudaProp.maxThreadsPerBlock);
```

will tell you the maximum number of threads allowed per block (1024 on Fermi and Kepler).

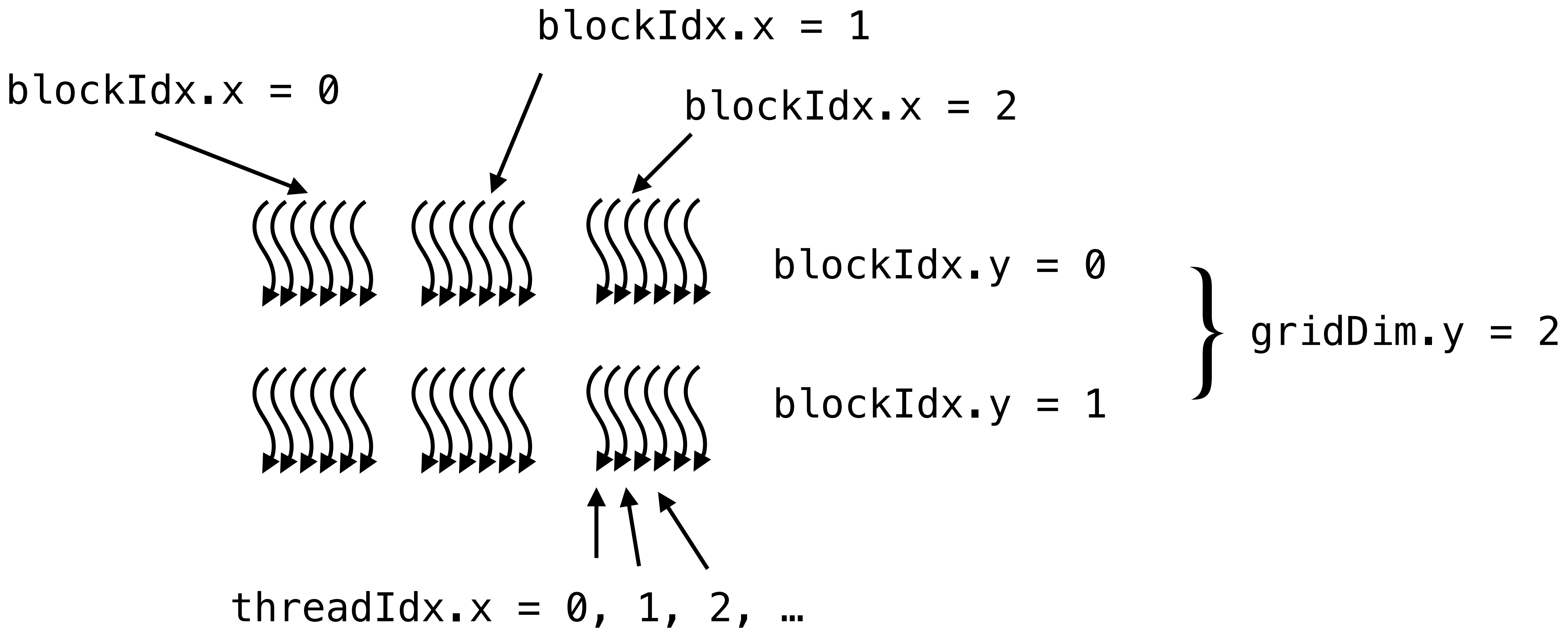
the maximum number of blocks (per dimension) is 65535

vector addition

a much better way to launch the kernel:

```
// threads per block should be multiple of the warp
// size (32) and has max value cudaProp.maxThreadsPerBlock
int threads_per_block = 128;
int maxblocks         = 65535;

int nblocks_x = n / threads_per_block;
int nblocks_y = 1;

if ( n % threads_per_block != 0 ) {
   nblocks_x = (n+threads_per_block-n % threads_per_block )/threads_per_block;
}

if (nblocks_x > maxblocks){
   nblocks_y = nblocks_x / maxblocks + 1;
   nblocks_x = nblocks_x / nblocks_y + 1;
}

// a two-dimensional grid: nblocks_x by nblocks_y
dim3 dimgrid (nblocks_x,nblocks_y);

vecadd_gpu_2d_grid<<<dimgrid,threads_per_block>>>(n,aGPU,bGPU,cGPU);
```

vector addition



```
// C = A + B, evaluated on a 2-dimensional grid of blocks, each with
// at least one thread.
__global__ void vecadd_gpu_2d_grid(int n, double * a, double * b, double * c) {

    int blockid = blockIdx.x*gridDim.y + blockIdx.y;
    int id      = blockid*blockDim.x + threadIdx.x;
    if ( id >= n ) return;

    c[id] = a[id] + b[id];
}
```

note that we now must check if the thread id falls within the bounds of our array

vector addition

lastly, we need to copy the result back from the device

```
// copy result back to host from device:
cudaMemcpy(c,cGPU,n*sizeof(double),cudaMemcpyDeviceToHost);
```

note the direction is different this time

vector addition

timing your functions:

```
double start = omp_get_wtime();

for (int i = 0; i < 1000; i++) {
    vecadd_gpu_2d_grid<<<dimgrid,threads_per_block>>>(n,aGPU,bGPU,cGPU);
}
// don't forget to block the CPU code from proceding
cudaThreadSynchronize();

double end = omp_get_wtime();
double gputime = end-start;
```

so, how much faster is the GPU than the CPU for this kernel?

what if you include the memory transfer in the timings?