Super Instruction Architecture for Heterogeneous Systems

Victor Lotric, Nakul Jindal, Erik Deumens, Rod Bartlett, Beverly Sanders

UF UNIVERSITY of FLORIDA

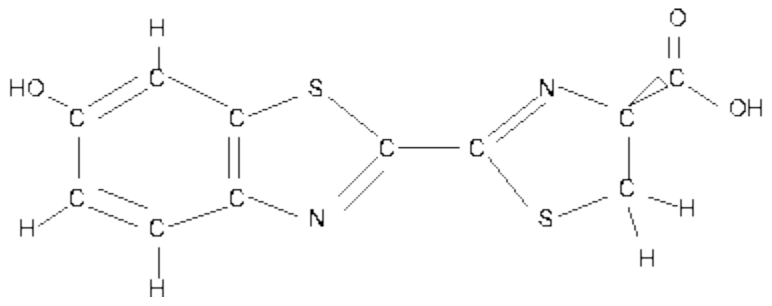
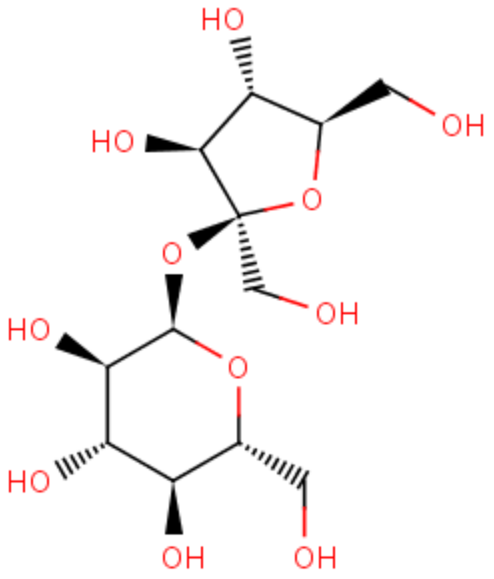
Super Instruction Architecture

Motivated by Computational Chemistry—Coupled Cluster Methods

- Dominated by tensor algebra using very large, dense multi-dimensional arrays
- Irregular access patterns
- Very complex algorithms--need abstraction at level that supports experimentation with algorithms

ACES III

- www.qtp.ufl.edu/ACES

Program characteristics

- Typical data requirements
  - CCSD for 100 electrons
    - 2-10 ~80 GB arrays
    - One ~800GB array

Program characteristics

- Typical data requirements
  - CCSD for 100 electrons
    - 2-10 ~80 GB arrays
    - One ~800GB array

Need to be distributed.
Some stored on disk

Program characteristics

- Typical data requirements
  - CCSD for 100 electrons
    - 2-10 ~80 GB arrays
    - One ~800GB array

Not stored, (re) compute as needed

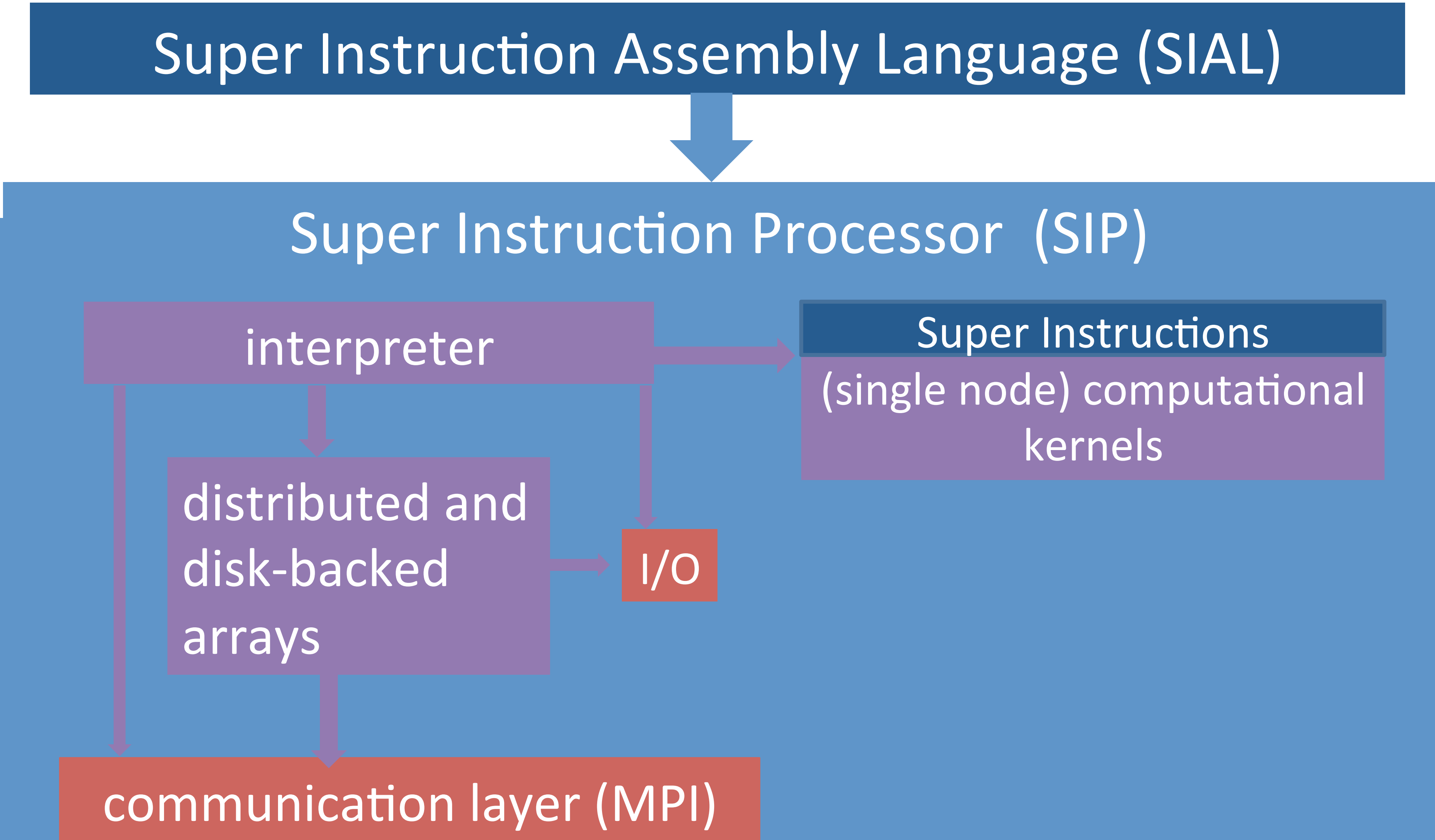
- Complex tradeoffs between
  - Computation
  - Memory usage
  - Communication

One term from the coupled cluster model*

```
hbar[a,b,i,j] == sum[f[b,c] * t[i,j,a,c], c] - sum[f[k,c] * t[k,b] * t[i,j,a,c], k,c] + sum[f[a,c] * t[i,j,c,b], c] - sum[f[k,c] * t[k,a] * t[i,j,c,b], k,c] - sum[f[k,j] * t[i,k,a,b], k] - sum[f[k,c] *
t[j,c] * t[i,k,a,b], k,c] - sum[f[k,i] * t[j,k,b,a], k] - sum[f[k,c] * t[i,c] * t[j,k,b,a], k,c] + sum[t[i,c] * t[j,d] * v[a,b,c,d], c,d] + sum[t[i,j,c,d] * v[a,b,c,d], c,d] + sum[t[j,c] * v[a,b,i,c],
c] - sum[t[k,b] * v[a,k,i,j], k] + sum[t[i,c] * v[b,a,j,c], c] - sum[t[k,a] * v[b,k,j,i], k] - sum[t[k,d] * t[i,j,c,b] * v[k,a,c,d], k,c,d] - sum[t[i,c] * t[j,k,b,d] * v[k,a,c,d], k,c,d] - sum[t[j,c]
* t[k,b] * v[k,a,c,i], k,c] + 2 * sum[t[j,k,b,c] * v[k,a,c,i], k,c] - sum[t[j,k,c,b] * v[k,a,c,i], k,c] - sum[t[i,c] * t[j,d] * t[k,b] * v[k,a,d,c], k,c,d] + 2 * sum[t[k,d] * t[i,j,c,b] * v[k,a,d,c],
k,c,d] - sum[t[k,b] * t[i,j,c,d] * v[k,a,d,c], k,c,d] - sum[t[j,d] * t[i,k,c,b] * v[k,a,d,c], k,c,d] + 2 * sum[t[i,c] * t[j,k,b,d] * v[k,a,d,c], k,c,d] - sum[t[i,c] * t[j,k,d,b] * v[k,a,d,c], k,c,d] -
sum[t[j,k,b,c] * v[k,a,i,c], k,c] - sum[t[i,c] * t[k,b] * v[k,a,j,c], k,c] - sum[t[i,k,c,b] * v[k,a,j,c], k,c] - sum[t[i,c] * t[j,d] * t[k,a] * v[k,b,c,d], k,c,d] - sum[t[k,d] * t[i,j,a,c] * v[k,b,c,d],
k,c,d] - sum[t[k,a] * t[i,j,c,d] * v[k,b,c,d], k,c,d] + 2 * sum[t[j,d] * t[i,k,a,c] * v[k,b,c,d], k,c,d] - sum[t[j,d] * t[i,k,c,a] * v[k,b,c,d], k,c,d] - sum[t[i,c] * t[j,k,d,a] * v[k,b,c,d], k,c,d]
- sum[t[i,c] * t[k,a] * v[k,b,c,j], k,c] + 2 * sum[t[i,k,a,c] * v[k,b,c,j], k,c] - sum[t[i,k,c,a] * v[k,b,c,j], k,c] + 2 * sum[t[k,d] * t[i,j,a,c] * v[k,b,d,c], k,c,d] - sum[t[j,d] * t[i,k,a,c] *
v[k,b,d,c], k,c,d] - sum[t[j,c] * t[k,a] * v[k,b,i,c], k,c] - sum[t[j,k,c,a] * v[k,b,i,c], k,c] - sum[t[i,k,a,c] * v[k,b,j,c], k,c] + sum[t[i,c] * t[j,d] * t[k,a] * t[l,b] * v[k,l,c,d], k,l,c,d] - 2 *
sum[t[k,b] * t[l,d] * t[i,j,a,c] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[k,a] * t[l,d] * t[i,j,c,b] * v[k,l,c,d], k,l,c,d] + sum[t[k,a] * t[l,b] * t[i,j,c,d] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[j,c] * t[l,d] *
t[i,k,a,b] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[j,d] * t[l,b] * t[i,k,a,c] * v[k,l,c,d], k,l,c,d] + sum[t[j,d] * t[l,b] * t[i,k,c,a] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,c] * t[l,d] * t[j,k,b,a] * v[k,l,c,d],
k,l,c,d] + sum[t[i,c] * t[l,a] * t[j,k,b,d] * v[k,l,c,d], k,l,c,d] + sum[t[i,c] * t[l,b] * t[j,k,d,a] * v[k,l,c,d], k,l,c,d] + sum[t[i,k,c,d] * t[j,l,b,a] * v[k,l,c,d], k,l,c,d] + 4 * sum[t[i,k,a,c] *
t[j,l,b,d] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,k,c,a] * t[j,l,b,d] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,k,a,b] * t[j,l,c,d] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,k,a,c] * t[j,l,d,b] * v[k,l,c,d], k,l,c,d]
+ sum[t[i,k,c,a] * t[j,l,d,b] * v[k,l,c,d], k,l,c,d] + sum[t[i,c] * t[j,d] * t[k,l,a,b] * v[k,l,c,d], k,l,c,d] + sum[t[i,j,c,d] * t[k,l,a,b] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,j,c,b] * t[k,l,a,d] *
v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,j,a,c] * t[k,l,b,d] * v[k,l,c,d], k,l,c,d] + sum[t[j,c] * t[k,b] * t[l,a] * v[k,l,c,i], k,l,c] + sum[t[l,c] * t[j,k,b,a] * v[k,l,c,i], k,l,c] - 2 * sum[t[l,a] * t[j,k,b,c]
* v[k,l,c,i], k,l,c] + sum[t[l,a] * t[j,k,c,b] * v[k,l,c,i], k,l,c] - 2 * sum[t[k,c] * t[j,l,b,a] * v[k,l,c,i], k,l,c] + sum[t[k,a] * t[j,l,b,c] * v[k,l,c,i], k,l,c] + sum[t[k,b] * t[j,l,c,a] * v[k,l,c,i],
k,l,c] + sum[t[j,c] * t[l,k,a,b] * v[k,l,c,i], k,l,c] + sum[t[i,c] * t[k,a] * t[l,b] * v[k,l,c,j], k,l,c] + sum[t[l,c] * t[i,k,a,b] * v[k,l,c,j], k,l,c] - 2 * sum[t[l,b] * t[i,k,a,c] * v[k,l,c,j], k,l,c]
+ sum[t[l,b] * t[i,k,c,a] * v[k,l,c,j], k,l,c] + sum[t[i,c] * t[k,l,a,b] * v[k,l,c,j], k,l,c] + sum[t[j,c] * t[l,d] * t[i,k,a,b] * v[k,l,d,c], k,l,c,d] + sum[t[j,d] * t[l,b] * t[i,k,a,c] * v[k,l,d,c],
k,l,c,d] + sum[t[j,d] * t[l,a] * t[i,k,c,b] * v[k,l,d,c], k,l,c,d] - 2 * sum[t[i,k,c,d] * t[j,l,b,a] * v[k,l,d,c], k,l,c,d] - 2 * sum[t[i,k,a,c] * t[j,l,b,d] * v[k,l,d,c], k,l,c,d] + sum[t[i,k,c,a] *
t[j,l,b,d] * v[k,l,d,c], k,l,c,d] + sum[t[i,k,a,b] * t[j,l,c,d] * v[k,l,d,c], k,l,c,d] + sum[t[i,k,c,b] * t[j,l,d,a] * v[k,l,d,c], k,l,c,d] + sum[t[i,k,a,c] * t[j,l,d,b] * v[k,l,d,c], k,l,c,d] + sum[t[k,a]
* t[l,b] * v[k,l,i,j], k,l] + sum[t[k,l,a,b] * v[k,l,i,j], k,l] + sum[t[k,b] * t[l,d] * t[i,j,a,c] * v[l,k,c,d], k,l,c,d] + sum[t[k,a] * t[l,d] * t[i,j,c,b] * v[l,k,c,d], k,l,c,d] + sum[t[i,c] * t[l,d] *
t[j,k,b,a] * v[l,k,c,d], k,l,c,d] - 2 * sum[t[i,c] * t[l,a] * t[j,k,b,d] * v[l,k,c,d], k,l,c,d] + sum[t[i,c] * t[l,a] * t[j,k,d,b] * v[l,k,c,d], k,l,c,d] + sum[t[i,j,c,b] * t[k,l,a,d] * v[l,k,c,d], k,l,c,d]
+ sum[t[i,j,a,c] * t[k,l,b,d] * v[l,k,c,d], k,l,c,d] - 2 * sum[t[l,c] * t[i,k,a,b] * v[l,k,c,j], k,l,c] + sum[t[l,b] * t[i,k,a,c] * v[l,k,c,j], k,l,c] + sum[t[l,a] * t[i,k,c,b] * v[l,k,c,j], k,l,c] +
v[a,b,i,j]
```

* Baumgartner, et al. *Proceedings of the IEEE*, 93:2 pp276-292.

Super Instruction Architecture


Super Instruction Assembly Language (SIAL)
Super Instruction Processor (SIP)
interpreter
Super Instructions
(single node) computational kernels
distributed and disk-backed arrays
I/O
communication layer (MPI)

Super Instructions and Super Numbers

- Traditional programming languages
  - unit of data: floating point number
  - operations: combine floating point numbers
  - but operations and data must be aggregated for good performance
- SIAL
  - unit of data: block (super number) of floating point numbers
  - operations: super instructions
    - Single node computational kernels
      - No communication
    - Written in a general purpose programming language
    - Some built-in, some implemented by domain experts

Algorithms in SIAL are expressed in terms of blocks and super instructions

Example: tensor contraction term

$$R^{\mu\nu}_{ij} = \sum_{\lambda\sigma} V^{\mu\nu}_{\lambda\sigma} T^{\lambda\sigma}_{ij}$$

Mathematical expression

F

Example: blocked version

$$R_{ij}^{\mu\nu} = \sum_{\lambda\sigma} V_{ij}^{\mu\nu} T_{ij}^{\lambda\sigma}$$

$$R(M,N,I,J)_{ij}^{\mu\nu} = \sum_{LS} \sum_{\lambda\in L} \sum_{\sigma\in S} \overset{F}{V}(M,N,L,S)_{\lambda\sigma}^{\mu\nu} T(L,S,I,J)_{ij}^{\lambda\sigma}$$

- Divide each dimension into segments
- *M,N,L,S,I,J* index segments of size *seg*
- Each block *R(M,N,I,J)* is a 4-index array of *seg*^{4} elements

Example: contraction super instruction

$$R^{\mu\nu}_{ij} = \sum_{\lambda\sigma} V^{\mu\nu}_{ij} T^{\lambda\sigma}_{ij}$$

SIAL programmer thinks about algorithm like this

$$R \quad {}^{\nu} = \sum_{LS} \sum_{\lambda\in L} \sum_{\sigma\in S} \overset{F}{V}(M,N,L,S)^{\mu\nu}_{\lambda\sigma} T(L,S,I,J)^{\lambda\sigma}_{ij}$$

$$R(M,N,I,J)^{\mu\nu}_{ij} = \sum_{LS} V(M,N,L,S)*T(L,S,I,J)$$

built-in super instruction

Super Instructions

- Single node computational kernels
  - No communication
  - CPU or GPU
- Written in a general purpose programming language, i.e. Fortran, C/C++ or CUDA
- Some built-in, some implemented by domain experts

Runtime System: SIP

- Organization
  - master
  - set of worker nodes
    - distributed array blocks managed by workers
  - set of I/O nodes that handle served (disk-backed arrays)
  - workers loop over op table containing SIAL byte code
    - byte code instruction corresponds to statement in SIAL
    - program structure available to runtime in convenient form
    - can profile instructions with minimal overhead
  - periodically checks for MPI messages
  - Implemented and tuned by parallel programing expert

Data Management

- Handles distributed data layout
  - data access very irregular
  - currently no attempts to exploit locality or block ownership

- Memory management at individual nodes “knows” about blocks and segment size for run (analysis during initialization)

- Blocks are automatically cached.

Benefits of the SIA with Current Systems

- Programmer productivity
  - Right abstraction
- Excellent parallel performance
  - Right granularity
- Easy to port, easy to tune*
  - Port SIP, SIAL programs still run
- Enables interesting code analyses
  - Predict memory usage during program initialization
  - Auto-generated performance models using inputs and timing info for super instructions
    - Predict run time and efficiency
    - Resource allocation and scheduling

*except Blue Gene

Goal: Extend the SIA to exploit GPUs

- Consequences of Architecture of SIA
  - Data is handled at a granularity that can be efficiently moved between nodes.
  - Computation steps will be time consuming enough for the runtime system to be able to effectively and automatically overlap communication and computation.
- Promise for exploiting GPUs
  - The computation is already partitioned into tasks that map conveniently onto CUDA kernels
  - Most super instructions lend themselves to straightforward data parallel implementations.

Parallel Implementation in SIAL

```
pardo M,N,I,J
    tmpsum(M,N,I,J) = 0.0
    do L
    do S
        get T(L,S,I,J)
        execute compute_integrals V(M,N,L,S)
        tmp(M,N,I,J) = V(M,N,L,S) * T(L,S,I,J)
        tmpsum(M,N,I,J) += tmp(M,N,I,J)
    enddo S
    enddo L
    put R(M,N,I,J) = tmpsum(M,N,I,J)
endpardo M,N,I,J
sip_barrier
```

Variable declarations and instantiation not shown

T and R are distributed arrays

Implementation in SIAL

```
pardo M,N,I,J
    tmpsum(M,N,I,J) = 0.0
    do L
    do S
        get T(L,S,I,J)
        execute compute_integrals V(M,N,L,S)
        tmp(M,N,I,J) = V(M,N,L,S) * T(L,S,I,J)
        tmpsum(M,N,I,J) += tmp(M,N,I,J)
    enddo S
    enddo L
    put R(M,N,I,J) = tmpsum(M,N,I,J)
endpardo M,N,I,J
sip_barrier
```

Divide iteration space among available workers and execute in parallel.

M,N,I,J,L,S count segments, have been defined earlier with symbolic ranges.

Implementation in SIAL

```
pardo M,N,I,J
    tmpsum(M,N,I,J) = 0.0
    do L
    do S
        get T(L,S,I,J)
        execute  compute_integrals V(M,N,L,S)
        tmp(M,N,I,J) = V(M,N,L,S) * T(L,S,I,J)
        tmpsum(M,N,I,J) += tmp(M,N,I,J)
    enddo S
    enddo L
    put R(M,N,I,J) = tmpsum(M,N,I,J)
endpardo M,N,I,J
sip_barrier
```

Request block of distributed array

Communication is one-sided and asynchronous

Implementation in SIAL

```
pardo M,N,I,J
    tmpsum(M,N,I,J) = 0.0
    do L
    do S
        get T(L,S,I,J)
        execute compute_integrals V(M,N,L,S)
        tmp(M,N,I,J) = V(M,N,L,S) * T(L,S,I,J)
        tmpsum(M,N,I,J) += tmp(M,N,I,J)
    enddo S
    enddo L
    put R(M,N,I,J) = tmpsum(M,N,I,J)
endpardo M,N,I,J
sip_barrier
```

Contraction.

Right hand side is implicit future—waits for T(L,S,I,J) if necessary

Implementation in SIAL

```
pardo M,N,I,J
    tmpsum(M,N,I,J) = 0.0
    do L
    do S
        get **T(L,S,I,J)**
        execute compute_integrals V(M,N,L,S)
        tmp(M,N,I,J) = V(M,N,L,S) * T(L,S,I,J)
        tmpsum(M,N,I,J) += tmp(M,N,I,J)
    enddo S
    enddo L
    put R(M,N,I,J) = tmpsum(M,N,I,J)
endpardo M,N,I,J
sip_barrier
```

Sends result to home node for block

Communication is one-sided and asynchronous

Enabling GPUs

- Attempt 1
  - Transparent to programmer and user
  - Contractions
  - Permutations
    - Y1ab(nu,j,mu,i) = Yab(mu,i,nu,j)
  - Within super instruction
    - If GPU available
      - Copy blocks to GPU
      - performs computation
      - Copy result blocks to host

Attempt 1: GPU-enabled implementation in SIAL

```
pardo M,N,I,J
    tmpsum(M,N,I,J) = 0.0
    do L
    do S
        get T(L,S,I,J)
        execute compute_integrals V(M,N,L,S)
        tmp(M,N,I,J) = V(M,N,L,S) * T(L,S,I,J)
        tmpsum(M,N,I,J) += tmp(M,N,I,J)
    enddo S
    enddo L
    put R(M,N,I,J) = tmpsum(M,N,I,J)
endpardo M,N,I,J
sip_barrier
```

No change to SIAL

SIP executes on GPU if one is available

Results from first attempt

- Good speedup for contractions alone
- Disappointing improvement for overall computation
  - Hurry up and wait (for needed block)
  - Unnecessary data transfer

GPU Directives

Directive	Description
set_gpu_on	Execute following instructions on GPU if one is available
set_gpu_off	Execute following instructions on CPU
load_gpu_input <block>	Allocate memory for <block> on GPU and copy data from host
unload_gpu_output <block>	Copy contents of block from GPU to host
allocate_gpu_block <block>	Allocate memory for <block> on GPU
free_gpu_block <block>	Free memory for <block> on GPU

Second approach

- Decouple data movement from CUDA super instruction implementations
- Programmer annotates computationally intensive parts of code

GPU-enabled implementation in SIAL with Directives

```
pardo M,N,I,J
     tmpsum(M,N,I,J) = 0.0
       set_gpu_on
       allocate_gpu_block  tmpsum(M,N,I,J)
       allocate_gpu_block  tmp(M,N,I,J)
      do L
             do S
                    get T(L,S,I,J)
                    compute_integrals V(M,N,L,S)
                    load_gpu_input  T(L,S,I,J)
                    load_gpu_input V(M,N,L,S)
                    tmp(M,N,I,J) = V(M,N,L,S) * T(L,S,I,J)
                    tmpsum(M,N,I,J) += tmp(M,N,I,J)
                    free_gpu_block V(M,N,L,S)
                    free_gpu_block T(L,S,I,J)
             enddo S
        enddo L
        unload_gpu_output tmpsum(M,N,I,J)
        put R(M,N,I,J) = tmpsum(M,N,I,J)
        free_gpu_block tmp(M,N,I,J)
        free_gpu_block tmpsum(M,N,I,J)
        set_gpu_off
endpardo M,N,I,J
```

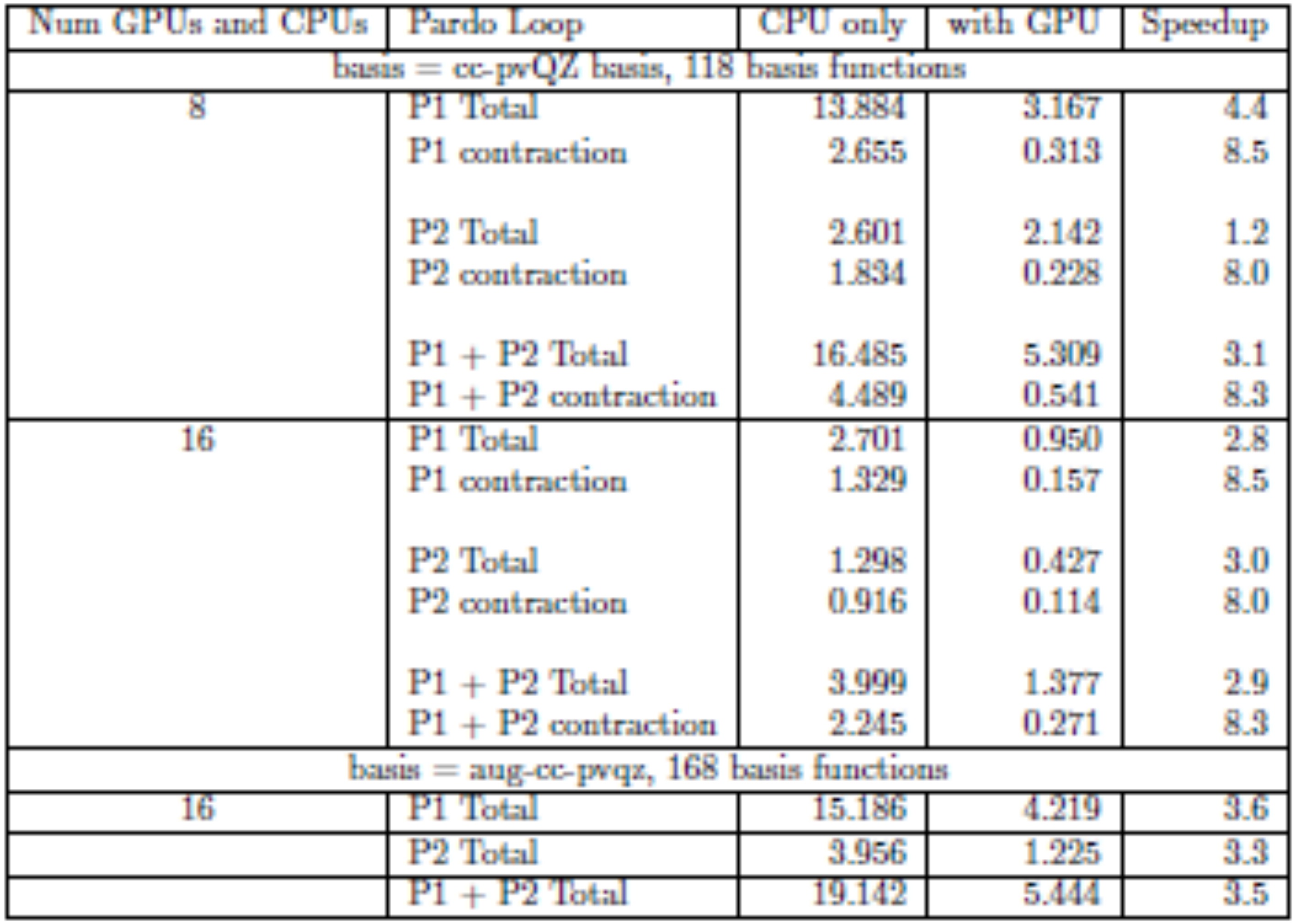
Green: directive
Red: GPU
Black: CPU

If no GPU available, everything executed on CPU with unchanged semantics

Performance Results

- Only use GPU for most computationally intense loops
- Programmer still needs to be mindful of overall structure of loop
- Next slide shows performance results for small system on UF HPC Center
  - Annotated two most computationally intense loops in CCSD calculation

Performance Results



Num GPUs and CPUs	Pardo Loop	CPU only	with GPU	Speedup
basis = cc-pwQZ basis, 118 basis functions				
8	P1 Total	13.884	3.167	4.4
	P1 contraction	2.655	0.313	8.5
	P2 Total	2.601	2.142	1.2
	P2 contraction	1.834	0.228	8.0
	P1 + P2 Total	16.485	5.309	3.1
	P1 + P2 contraction	4.489	0.541	8.3
16	P1 Total	2.701	0.950	2.8
	P1 contraction	1.329	0.157	8.5
	P2 Total	1.298	0.427	3.0
	P2 contraction	0.916	0.114	8.0
	P1 + P2 Total	3.999	1.377	2.9
	P1 + P2 contraction	2.245	0.271	8.3
basis = aug-cc-pwqz, 168 basis functions				
16	P1 Total	15.186	4.219	3.6
	P2 Total	3.956	1.225	3.3
	P1 + P2 Total	19.142	5.444	3.5

Annotation Burden?

- Only beneficial for most computationally intensive loops
- Programmer may need to rethink loop structure
  - (this probably improves the performance of the CPU code, too)
- Example on previous slides had low computation to annotation ratio.

Future Work

- The SIA has sophisticated analysis tools
  - Performance: SIPMap (IPDPS'13)
  - Memory: "dry run"
- Incorporate GPU knowledge into these analyses
- Provide more sophisticated analysis to help ease annotation burden

Extra slides

- Actual CCSD code

CCSD Parallel Loop

```
PARDO lambda, sigma
#allocate and initialize CPU memory, compute integral block on CPU
        allocate LTAO_ab(lambda,*,sigma,*)
        DO i
        DO j
                request  TAO_ab(lambda,i,sigma,j)
                LTAO_ab(lambda,i,sigma,j) = TAO_ab(lambda,i,sigma,j)
        ENDDO j
        ENDDO i
        DO mu
        DO nu
                WHERE mu < nu
                allocate LT2AO_ab1(mu,*,nu,*)
                allocate LT2AO_ab2(nu,*,mu,*)
                compute_integrals aoint(lambda,mu,sigma,nu)
```

```
#enable gpu, allocate an initialize blocks on GPU
set_gpu_on
load_gpu_input aoint(lambda,mu,sigma,nu)
#allocate and copy data from CPU
    DO i1
    DO j1
        load_gpu_input LT2AO_ab1(mu,i1,nu,j1)
         load_gpu_input LT2AO_ab2(nu,j1,mu,i1)
          load_gpu_input LTAO_ab(lambda,i1,sigma,j1)
     ENDDO j1
     ENDDO i1
```

```
DO i
    DO j
#perform computations on GPU
         Yab(mu,i,nu,j)  = 0.0
         Y1ab(nu,j,mu,i) = 0.0
          load_gpu_temp Yab(mu,i,nu,j)   #allocate temp blocks on GPU
           load_gpu_temp Y1ab(nu,j,mu,i)
           Yab(mu,i,nu,j) = aoint(lambda,mu,sigma,nu)*LTAO_ab
(lambda,i,sigma,j)
           Y1ab(nu,j,mu,i) = Yab(mu,i,nu,j) #permutation
           LT2AO_ab1(mu,i,nu,j) += Yab(mu,i,nu,j)  #elementwise sums
           LT2AO_ab2(nu,j,mu,i) += Y1ab(nu,j,mu,i) #elementwise sums
          free_gpu_block Yab(mu,i,nu,j) #free temp blocks on GPU
          free_gpu_block Y1ab(nu,j,mu,i)
ENDDO j
ENDDO i
```

```
#copy results to CPU , free blocks on GPU and disable
            DO i1
            DO j1
                  unload_gpu_block LT2AO_ab1(mu,i1,nu,j1)
                  unload_gpu_block LT2AO_ab2(nu,j1,mu,i1)
                  free_gpu_block LT2AO_ab1(mu,i1,nu,j1)
                  free_gpu_block LT2AO_ab2(nu,j1,mu,i1)
                  free_gpu_block LTAO_ab(lambda,i1,sigma,j1)
            ENDDO j1
            ENDDO i1
            free_gpu_block aoint(lambda,mu,sigma,nu)
            set_gpu_off
```

```
#finish on CPU
                        DO i
                        DO j
                                    prepare T2AO_ab(mu,i,nu,j) += LT2AO_ab1(mu,i,nu,j
                                    prepare T2AO_ab(nu,j,mu,i) += LT2AO_ab2(nu,j,mu,i)
                        ENDDO j
                        ENDDO i
                                    deallocate LT2AO_ab1(mu,*,nu,*)
#free local blocks on CPU
                                    deallocate LT2AO_ab2(nu,*,mu,*)
                        ENDDO nu
                        ENDDO mu
            deallocate LTAO_ab(lambda,*,sigma,*)
ENDPARDO lambda, sigma
```