A distributed-memory framework for tensor contractions

Edgar Solomonik

Department of EECS, Computer Science Division, UC Berkeley

Dec 9, 2013

Outline

Coupled Cluster as a hypergraph computation

- Graphs describe the connectivity of a set of vertices

Coupled Cluster as a hypergraph computation

- Graphs describe the connectivity of a set of vertices
    - an edge in a graph is a pair of vertices

Coupled Cluster as a hypergraph computation

- Graphs describe the connectivity of a set of vertices
  - an edge in a graph is a pair of vertices
  - an edge in a hypergraph is a set of vertices

Coupled Cluster as a hypergraph computation

- Graphs describe the connectivity of a set of vertices
  - an edge in a graph is a pair of vertices
  - an edge in a hypergraph is a set of vertices
- Coupled Cluster can be described in terms of a directed hypergraph $G = (V, E)$

Coupled Cluster as a hypergraph computation

- Graphs describe the connectivity of a set of vertices
  - an edge in a graph is a pair of vertices
  - an edge in a hypergraph is a set of vertices
- Coupled Cluster can be described in terms of a directed hypergraph $G = (V, E)$
  - the vertices $V = O \cup U$ contain electrons O and basis functions U

Coupled Cluster as a hypergraph computation

- Graphs describe the connectivity of a set of vertices
  - an edge in a graph is a pair of vertices
  - an edge in a hypergraph is a set of vertices
- Coupled Cluster can be described in terms of a directed hypergraph $G = (V, E)$
  - the vertices $V = O \cup U$ contain electrons O and basis functions U
  - one-electron integrals can be represented as regular edges

Coupled Cluster as a hypergraph computation

- Graphs describe the connectivity of a set of vertices
  - an edge in a graph is a pair of vertices
  - an edge in a hypergraph is a set of vertices
- Coupled Cluster can be described in terms of a directed hypergraph $G = (V, E)$
  - the vertices $V = O \cup U$ contain electrons O and basis functions U
  - one-electron integrals can be represented as regular edges
  - two-electron integrals can be represented as hypergraph edges $v^{pq}_{rs} \in (V \times V \times V \times V)$

Coupled Cluster as a hypergraph computation

- Graphs describe the connectivity of a set of vertices
  - an edge in a graph is a pair of vertices
  - an edge in a hypergraph is a set of vertices
- Coupled Cluster can be described in terms of a directed hypergraph $G = (V, E)$
  - the vertices $V = O \cup U$ contain electrons O and basis functions U
  - one-electron integrals can be represented as regular edges
  - two-electron integrals can be represented as hypergraph edges $v^{pq}_{rs} \in (V \times V \times V \times V)$
  - the amplitudes may also be represented as hypergraph edges (or paths) $t^{ab}_{ij} \in (O \times O \times \ldots) \rightarrow (U \times U \times \ldots)$

Coupled Cluster as a hypergraph computation

- Graphs describe the connectivity of a set of vertices
  - an edge in a graph is a pair of vertices
  - an edge in a hypergraph is a set of vertices
- Coupled Cluster can be described in terms of a directed hypergraph $G = (V, E)$
  - the vertices $V = O \cup U$ contain electrons O and basis functions U
  - one-electron integrals can be represented as regular edges
  - two-electron integrals can be represented as hypergraph edges $v^{pq}_{rs} \in (V \times V \times V \times V)$
  - the amplitudes may also be represented as hypergraph edges (or paths) $t^{ab}_{ij} \in (O \times O \times \ldots) \rightarrow (U \times U \times \ldots)$
  - Coupled Cluster iteratively updates the hypergraph 'paths' based on previously known best values

Motivation

Hypergraphs are represented numerically as tensors

- tensor symmetry is implicit from hypergraph edge definition
- Coupled Cluster is represented numerically as tensor contractions

Tensor contractions are a mathematical encoding of dependencies

- data and its structure is described as tensors
- interaction among data is described as tensor contractions
- general beyond Coupled Cluster (or even quantum chemistry)

Basic specification for a tensor library

A tensor contraction library should provide

- tensor objects that express structure
  - partial and full symmetry/anti-symmetry
  - sparsity
- user-level contractions defined by indices rather than loops
- data accessibility in multiple forms
  - full dense tensor
  - sparse index-value pairs
  - slice (subtensor)
  - subset of indices along each dimension

Specification for a distributed tensor library

In a distributed-memory tensor contraction library,

- tensor objects should live on a *any* set of processors (MPI comm)
- tensor data should be partitioned among (mapped onto) the processors internally
- tensors should be able to migrate between mappings
- the framework should select an algorithm and tensor mappings for each contraction
- it should be possible to schedule many contractions in parallel

Mapping in Cyclops Tensor Framework (CTF)

Decompose tensor into blocks (virtual processors) and map blocks onto processors

- map a tensor with edge lengths $(n_1, n_2, \ldots)$ tensor to a $(p_1, p_2, \ldots)$ via a $(v_1, v_2, \ldots)$ virtual topology, such that

Mapping in Cyclops Tensor Framework (CTF)

Decompose tensor into blocks (virtual processors) and map blocks onto processors

- map a tensor with edge lengths $(n_1, n_2, \ldots)$ tensor to a $(p_1, p_2, \ldots)$ via a $(v_1, v_2, \ldots)$ virtual topology, such that
  - $v_i = 0 \mod p_i$ for (enforce load balance)

Mapping in Cyclops Tensor Framework (CTF)

Decompose tensor into blocks (virtual processors) and map blocks onto processors

- map a tensor with edge lengths $(n_1, n_2, \ldots)$ tensor to a $(p_1, p_2, \ldots)$ via a $(v_1, v_2, \ldots)$ virtual topology, such that
  - $v_i = 0 \mod p_i$ for (enforce load balance)
  - $v_i = v_j$ if tensor dimensions i and j are symmetric (preserve symmetry)

Mapping in Cyclops Tensor Framework (CTF)

Decompose tensor into blocks (virtual processors) and map blocks onto processors

- map a tensor with edge lengths $(n_1, n_2, \ldots)$ tensor to a $(p_1, p_2, \ldots)$ via a $(v_1, v_2, \ldots)$ virtual topology, such that
  - $v_i = 0 \mod p_i$ for (enforce load balance)
  - $v_i = v_j$ if tensor dimensions i and j are symmetric (preserve symmetry)
  - typically want to maximize block size, $\prod_i n_i / v_i$

Mapping in Cyclops Tensor Framework (CTF)

Decompose tensor into blocks (virtual processors) and map blocks onto processors

- map a tensor with edge lengths $(n_1, n_2, \ldots)$ tensor to a $(p_1, p_2, \ldots)$ via a $(v_1, v_2, \ldots)$ virtual topology, such that
  - $v_i = 0 \mod p_i$ for (enforce load balance)
  - $v_i = v_j$ if tensor dimensions i and j are symmetric (preserve symmetry)
  - typically want to maximize block size, $\prod_i n_i/v_i$
- for each contraction, enforce new rules on mapping

Mapping in Cyclops Tensor Framework (CTF)

Decompose tensor into blocks (virtual processors) and map blocks onto processors

- map a tensor with edge lengths $(n_1, n_2, \ldots)$ tensor to a $(p_1, p_2, \ldots)$ via a $(v_1, v_2, \ldots)$ virtual topology, such that
  - $v_i = 0 \mod p_i$ for (enforce load balance)
  - $v_i = v_j$ if tensor dimensions i and j are symmetric (preserve symmetry)
  - typically want to maximize block size, $\prod_i n_i/v_i$
- for each contraction, enforce new rules on mapping
  - if two tensors share an index, mapped onto v_i in the first and onto v_j in the second, $v_i = v_j$

Mapping in Cyclops Tensor Framework (CTF)

Decompose tensor into blocks (virtual processors) and map blocks onto processors

- map a tensor with edge lengths $(n_1, n_2, \ldots)$ tensor to a $(p_1, p_2, \ldots)$ via a $(v_1, v_2, \ldots)$ virtual topology, such that
  - $v_i = 0 \mod p_i$ for (enforce load balance)
  - $v_i = v_j$ if tensor dimensions i and j are symmetric (preserve symmetry)
  - typically want to maximize block size, $\prod_i n_i/v_i$
- for each contraction, enforce new rules on mapping
  - if two tensors share an index, mapped onto v_i in the first and onto v_j in the second, $v_i = v_j$
  - etc...

The mapping process

Do in parallel over all physical topologies (foldings of the original torus)

1. map longest physical torus dimension to longest tensor dimension and repeat

The mapping process

Do in parallel over all physical topologies (foldings of the original torus)

1. map longest physical torus dimension to longest tensor dimension and repeat
2. select virtualization factors to preserve symmetry (as well as to match the algorithmic requirements)

The mapping process

Do in parallel over all physical topologies (foldings of the original torus)

1. map longest physical torus dimension to longest tensor dimension and repeat
2. select virtualization factors to preserve symmetry (as well as to match the algorithmic requirements)
3. calculate the necessary memory usage and communication cost of the algorithm

The mapping process

Do in parallel over all physical topologies (foldings of the original torus)

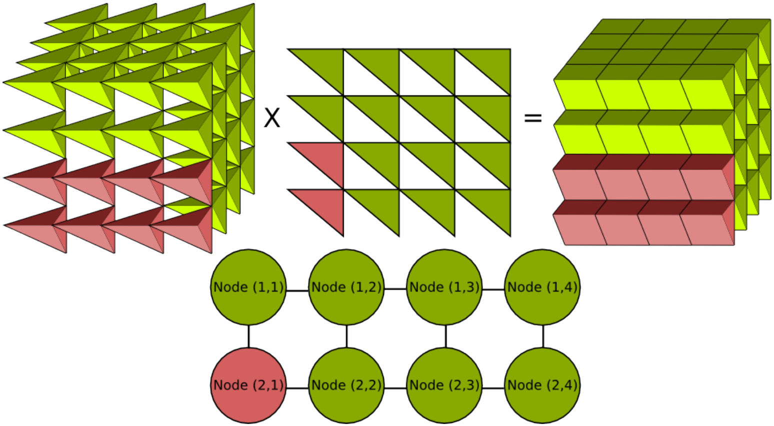
1. map longest physical torus dimension to longest tensor dimension and repeat
2. select virtualization factors to preserve symmetry (as well as to match the algorithmic requirements)
3. calculate the necessary memory usage and communication cost of the algorithm
4. consider whether and what type of redistribution is necessary for the mapping

The mapping process

Do in parallel over all physical topologies (foldings of the original torus)

1. map longest physical torus dimension to longest tensor dimension and repeat
2. select virtualization factors to preserve symmetry (as well as to match the algorithmic requirements)
3. calculate the necessary memory usage and communication cost of the algorithm
4. consider whether and what type of redistribution is necessary for the mapping
5. select the best mapping based on a performance model

3D tensor mapping



Redistribution in CTF

CTF provides three types of redistributions

- sparse index-value redistribution

Redistribution in CTF

CTF provides three types of redistributions

- sparse index-value redistribution
    - general but slow

Redistribution in CTF

CTF provides three types of redistributions

- sparse index-value redistribution
  - general but slow
  - easily accessible to user

Redistribution in CTF

CTF provides three types of redistributions

- sparse index-value redistribution
  - general but slow
  - easily accessible to user
- mapping-to-mapping redistribution

Redistribution in CTF

CTF provides three types of redistributions

- sparse index-value redistribution
  - general but slow
  - easily accessible to user
- mapping-to-mapping redistribution
  - allows a tensor to migrate from an ordered mapping to another

Redistribution in CTF

CTF provides three types of redistributions

- sparse index-value redistribution
  - general but slow
  - easily accessible to user
- mapping-to-mapping redistribution
  - allows a tensor to migrate from an ordered mapping to another
  - does not form indices explicitly (exploits global order)

Redistribution in CTF

CTF provides three types of redistributions

- sparse index-value redistribution
  - general but slow
  - easily accessible to user
- mapping-to-mapping redistribution
  - allows a tensor to migrate from an ordered mapping to another
  - does not form indices explicitly (exploits global order)
  - $\sim$10X faster than sparse redistribution

Redistribution in CTF

CTF provides three types of redistributions

- sparse index-value redistribution
  - general but slow
  - easily accessible to user
- mapping-to-mapping redistribution
  - allows a tensor to migrate from an ordered mapping to another
  - does not form indices explicitly (exploits global order)
  - $\sim$10X faster than sparse redistribution
- block-to-block redistribution

Redistribution in CTF

CTF provides three types of redistributions

- sparse index-value redistribution
  - general but slow
  - easily accessible to user
- mapping-to-mapping redistribution
  - allows a tensor to migrate from an ordered mapping to another
  - does not form indices explicitly (exploits global order)
  - $\sim$10X faster than sparse redistribution
- block-to-block redistribution
  - possible if the virtual decomposition (blocking) does not change

Redistribution in CTF

CTF provides three types of redistributions

- sparse index-value redistribution
  - general but slow
  - easily accessible to user
- mapping-to-mapping redistribution
  - allows a tensor to migrate from an ordered mapping to another
  - does not form indices explicitly (exploits global order)
  - $\sim$10X faster than sparse redistribution
- block-to-block redistribution
  - possible if the virtual decomposition (blocking) does not change
  - useful for reassigning physical dimensions

Redistribution in CTF

CTF provides three types of redistributions

- sparse index-value redistribution
  - general but slow
  - easily accessible to user
- mapping-to-mapping redistribution
  - allows a tensor to migrate from an ordered mapping to another
  - does not form indices explicitly (exploits global order)
  - $\sim$10X faster than sparse redistribution
- block-to-block redistribution
  - possible if the virtual decomposition (blocking) does not change
  - useful for reassigning physical dimensions
  - $\sim$10X faster than general mapping-to-mapping redistribution

Local transposition

Once the data is redistributed into the new mapping, we reorder it locally within blocks

- turns all non-symmetric block contractions into matrix multiplication

Local transposition

Once the data is redistributed into the new mapping, we reorder it locally within blocks

- turns all non-symmetric block contractions into matrix multiplication
- 'preserved' symmetries may be folded into one dimension, but broken ones cannot

Local transposition

Once the data is redistributed into the new mapping, we reorder it locally within blocks

- turns all non-symmetric block contractions into matrix multiplication
- 'preserved' symmetries may be folded into one dimension, but broken ones cannot
- maps dimensions which have symmetry that cannot be folded into matrix multiplication to have the longest stride

Local transposition

Once the data is redistributed into the new mapping, we reorder it locally within blocks

- turns all non-symmetric block contractions into matrix multiplication
- 'preserved' symmetries may be folded into one dimension, but broken ones cannot
- maps dimensions which have symmetry that cannot be folded into matrix multiplication to have the longest stride
- the contraction execution logic becomes

Local transposition

Once the data is redistributed into the new mapping, we reorder it locally within blocks

- turns all non-symmetric block contractions into matrix multiplication
- 'preserved' symmetries may be folded into one dimension, but broken ones cannot
- maps dimensions which have symmetry that cannot be folded into matrix multiplication to have the longest stride
- the contraction execution logic becomes
    1. nested SUMMA (distributed matrix multiplication)

Local transposition

Once the data is redistributed into the new mapping, we reorder it locally within blocks

- turns all non-symmetric block contractions into matrix multiplication
- 'preserved' symmetries may be folded into one dimension, but broken ones cannot
- maps dimensions which have symmetry that cannot be folded into matrix multiplication to have the longest stride
- the contraction execution logic becomes
  1. nested SUMMA (distributed matrix multiplication)
  2. nested call to iterate over virtual blocks

Local transposition

Once the data is redistributed into the new mapping, we reorder it locally within blocks

- turns all non-symmetric block contractions into matrix multiplication
- 'preserved' symmetries may be folded into one dimension, but broken ones cannot
- maps dimensions which have symmetry that cannot be folded into matrix multiplication to have the longest stride
- the contraction execution logic becomes
    1. nested SUMMA (distributed matrix multiplication)
    2. nested call to iterate over virtual blocks
    3. nested call to iterate over broken symmetric dimensions

Local transposition

Once the data is redistributed into the new mapping, we reorder it locally within blocks

- turns all non-symmetric block contractions into matrix multiplication
- 'preserved' symmetries may be folded into one dimension, but broken ones cannot
- maps dimensions which have symmetry that cannot be folded into matrix multiplication to have the longest stride
- the contraction execution logic becomes
    1. nested SUMMA (distributed matrix multiplication)
    2. nested call to iterate over virtual blocks
    3. nested call to iterate over broken symmetric dimensions
    4. nested call to DGEMM (matrix multiplication)

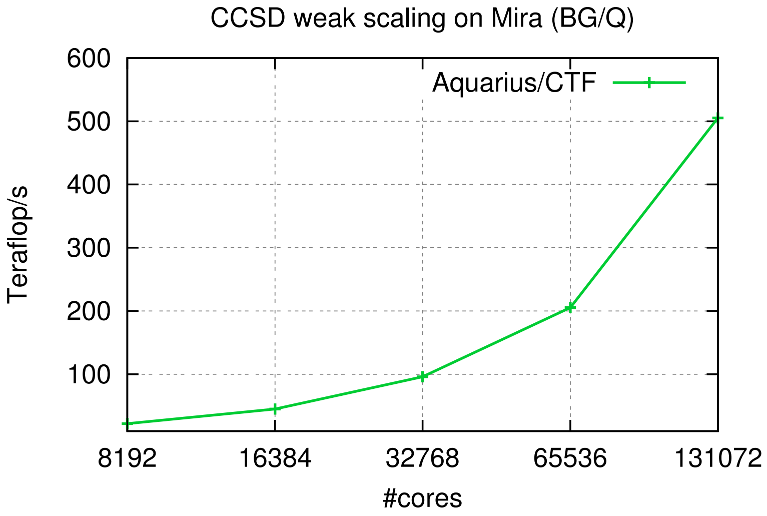
Comparison with NWChem on Cray XE6

CCSD iteration time on 64 nodes of Hopper:

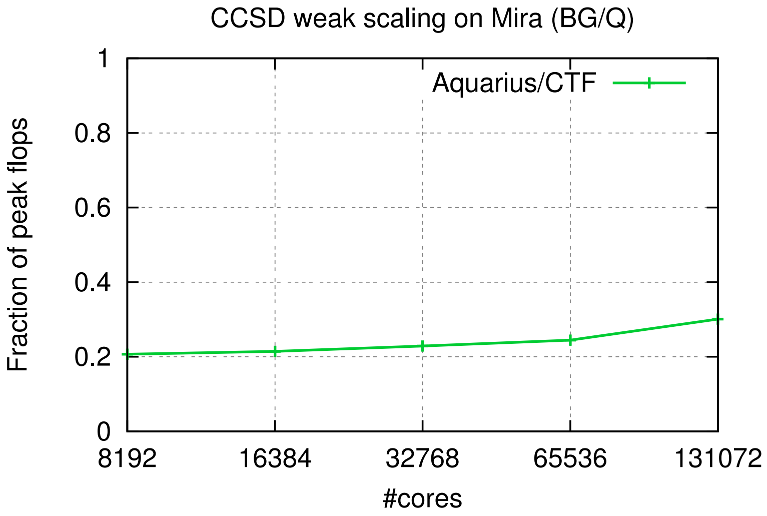
system	# electrons	# orbitals	CTF	NWChem
w5	25	205	14 sec	36 sec
w7	35	287	90 sec	178 sec
w9	45	369	127 sec	-
w12	60	492	336 sec	-

On 128 nodes, NWChem completed w9 in 223 sec, CTF in 73 sec.

Blue Gene/Q CTF/Aquarius CCSD up to 1250 orbitals, 250 electrons



Coupled Cluster efficiency on Blue Gene/Q



Problems posed by CCSDT

Tensor symmetry

- T3 amplitude tensors are symmetric up to 36 index permutations
- packing/unpacking requires many transpositions
- performing each permutation requires many contractions

Lots of contractions

- many contractions involve small tensors
- even the large contractions involve at least one 'smaller' tensor

CTF renovations for CCSDT

Much optimization to transposition kernels has been done

- new optimizations for mapping-to-mapping redistribution kernel (thanks to Devin)
- block-to-block redistribution introduced
- transpose and redistribution threaded with consideration for symmetric structure

Unpacking, repacking, and replication cause memory fragmentation

- cannot let tensors run free in the wild
- assign 'home' buffer (initial mapping) and migrate data back to it
- use internal stack for efficient large memory allocation management

Preliminary CCSDT results

Largest CTF/Aquarius CCSDT run so far

- 8 water molecules (40 electrons), cc-pVDZ basis set (192 atomic orbitals)
- done on 2048 nodes of BG/Q (128K cores)
- 15 mins per CCSDT iteration, ∼30 Teraflops, 23% time in dgemm

Preliminary comparison with NWChem for CCSDT on 32 nodes Hopper (iteration time)

- 3-waters, cc-pVDZ: CTF 100 sec, NWChem 160 sec
- 4-waters, cc-pVDZ: CTF 382 sec, NWChem 750 sec

Is CTF optimal?

- good question...

Is CTF optimal?

- good question...
- no!

Is CTF optimal?

- good question...
- no!
- why?

Symmetric tensor contractions via fully symmetric intermediates

- Let **b** be a vector of length n with elements

Symmetric tensor contractions via fully symmetric intermediates

- Let **b** be a vector of length *n* with elements
- Let **A** be a *n*-by-*n* symmetric matrix with elements

$$A_{ij} = A_{ji}$$

Symmetric tensor contractions via fully symmetric intermediates

- Let **b** be a vector of length n with elements
- Let **A** be a n-by-n symmetric matrix with elements

$$A_{ij} = A_{ji}$$

- Typically, we say the symmetry of **A** is broken and compute

$$c_i = \sum_{j=1}^{n} A_{ij} b_j \tag{1}$$

Symmetric tensor contractions via fully symmetric intermediates

- Let **b** be a vector of length *n* with elements
- Let **A** be a *n*-by-*n* symmetric matrix with elements

$$A_{ij} = A_{ji}$$

- Typically, we say the symmetry of **A** is broken and compute

$$c_i = \sum_{j=1}^{n} A_{ij} b_j \tag{1}$$

- Instead we can use half the number of multiplications

$$c_i = \sum_{j=1}^{n} A_{ij} \cdot (b_i + b_j) - \left(\sum_{j=1}^{n} A_{ij} \right) b_i$$

Symmetric tensor contractions via fully symmetric intermediates

- Let **b** be a vector of length *n* with elements
- Let **A** be a *n*-by-*n* symmetric matrix with elements

$$A_{ij} = A_{ji}$$

- Typically, we say the symmetry of **A** is broken and compute

$$c_i = \sum_{j=1}^{n} A_{ij} b_j \tag{1}$$

- Instead we can use half the number of multiplications

$$c_i = \sum_{j=1}^{n} A_{ij} \cdot (b_i + b_j) - \left(\sum_{j=1}^{n} A_{ij} \right) b_i$$

- A similar reorganization is possible for the symmetrized outer product

General fast symmetric tensor contractions

Given *fully* symmetric **A**, **B**, and **C**, compute

$$C_{i_1\dots i_{s+t}} = \sum_{((j_1\dots j_s),(l_1\dots l_t))\in\chi_s(i_1\dots i_{s+t})} \left(\sum_{k_1\dots k_v} A^{k_1\dots k_v}_{j_1\dots j_s} \cdot B^{l_1\dots l_t}_{k_1\dots k_v} \right).$$

Typically computed by (implicitly) forming partially-symmetric $\bar{\mathbf{C}}$

$$\bar{C}^{l_1\dots l_t}_{j_1\dots j_s} = \sum_{k_1\dots k_v} A^{k_1\dots k_v}_{j_1\dots j_s} \cdot B^{l_1\dots l_t}_{k_1\dots k_v}.$$

Cost is $\frac{n^{s+t+v}}{s!t!v!}$, via fully symmetric intermediates it becomes,

$$\binom{n}{s+t+v} \approx \frac{n^{s+t+v}}{(s+t+v)!}$$

Summary

Cyclops Tensor Framework (CTF)

- `ctf.cs.berkeley.edu`, BSD license, try it, use it
- stand-alone library requiring only MPI+OpenMP+BLAS
- Tested on gcc/intel/xlc, Mira/Carver/Hopper/Edison/Apple
- High performance algebra for multidimensional symmetric arrays
- In its essence, CTF is a library for mapping and communication orchestration of data via mathematical user-level language (operators)
- Its not optimal, because there are faster algorithms for symmetric contractions (but the software abstractions are still correct!)

Future and ongoing work

Cyclops Tensor Framework

- scheduling and concurrent execution of contractions
- better internal performance models
- exposure of a mapping interface to the user
- sparse tensors
- software realization of fast symmetric tensor contraction algorithms

Collaborators and acknowledgements

Collaborators:

- Devin Matthews, UT Austin (contributions to CTF, teaching me CC, and development of Aquarius on top of CTF)
- Jeff Hammond, Argonne National Laboratory (initiated project and provides continuing advice)
- James Demmel and Kathy Yelick, UC Berkeley (high-level advising)

Grants:

- Krell DOE Computational Science Graduate Fellowship

Backup slides