CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

AI CENTER

Paralelní a distribuované výpočty (B4B36PDV)

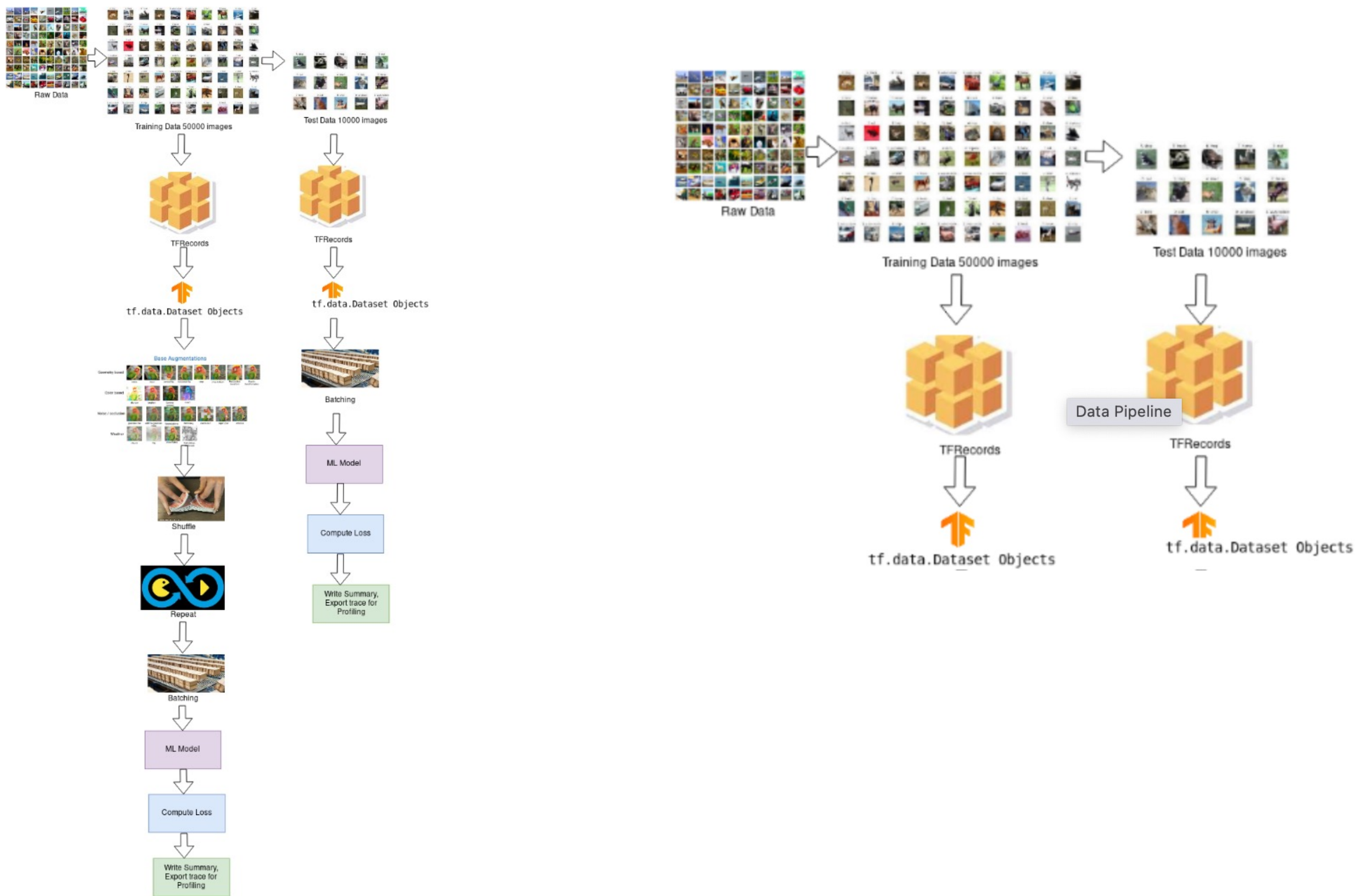
Jakub Mareček, Michal Jakob

jakub.marecek@fel.cvut.cz

Artificial Intelligence Center
Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague

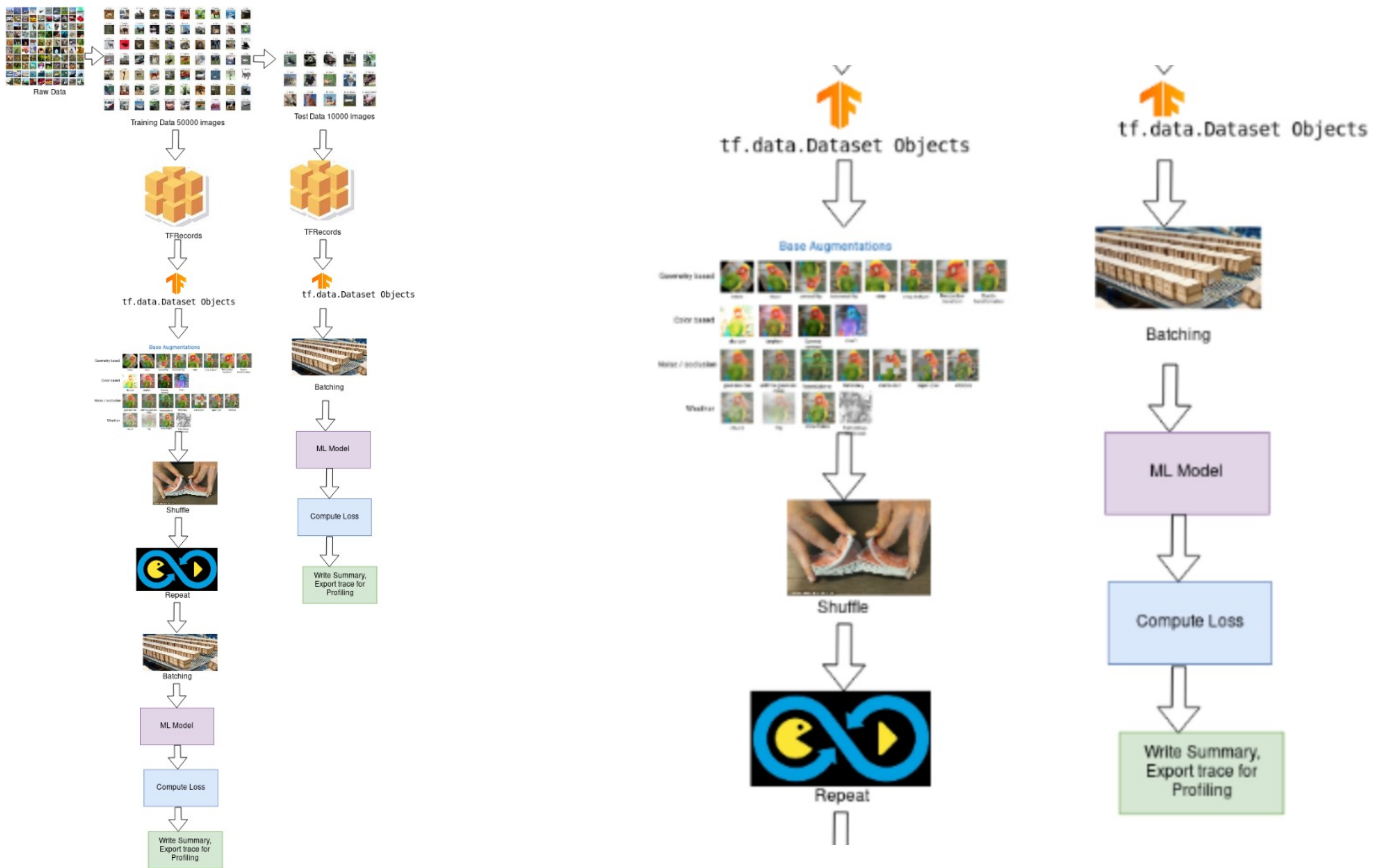
Dnes s využitím materiálů Brana Bošanského: https://cw.fel.cvut.cz/b192/courses/b4b36pdv/lectures/start

Motivation



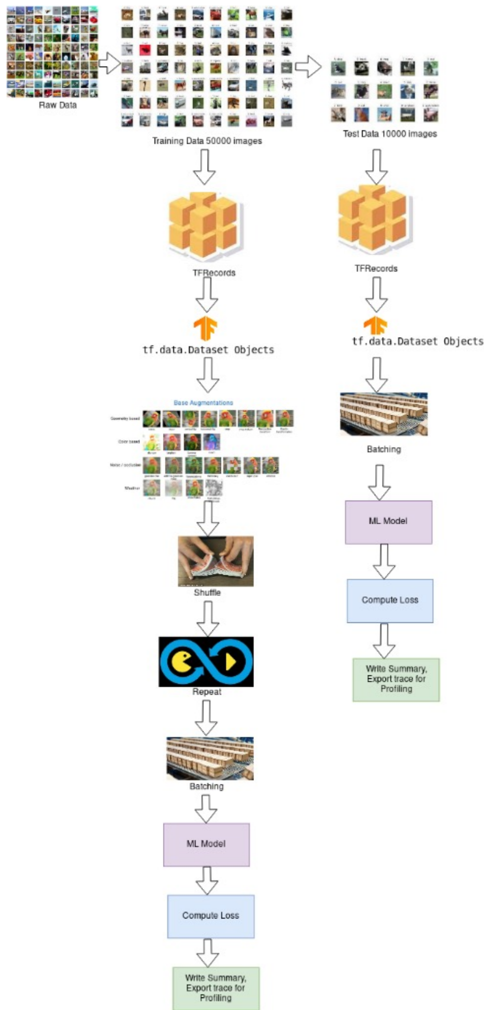
https://www.pluralsight.com/guides/time-profiling-neural-networks-model

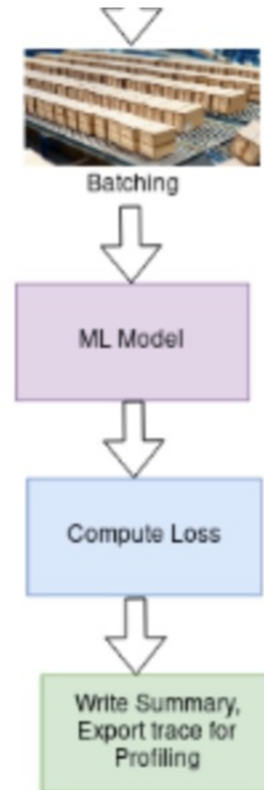
Motivation



https://www.pluralsight.com/guides/time-profiling-neural-networks-model

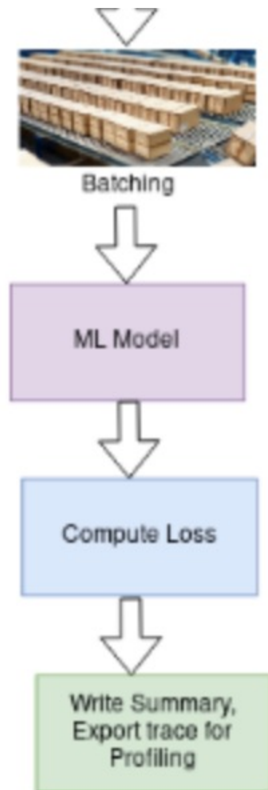
Motivation

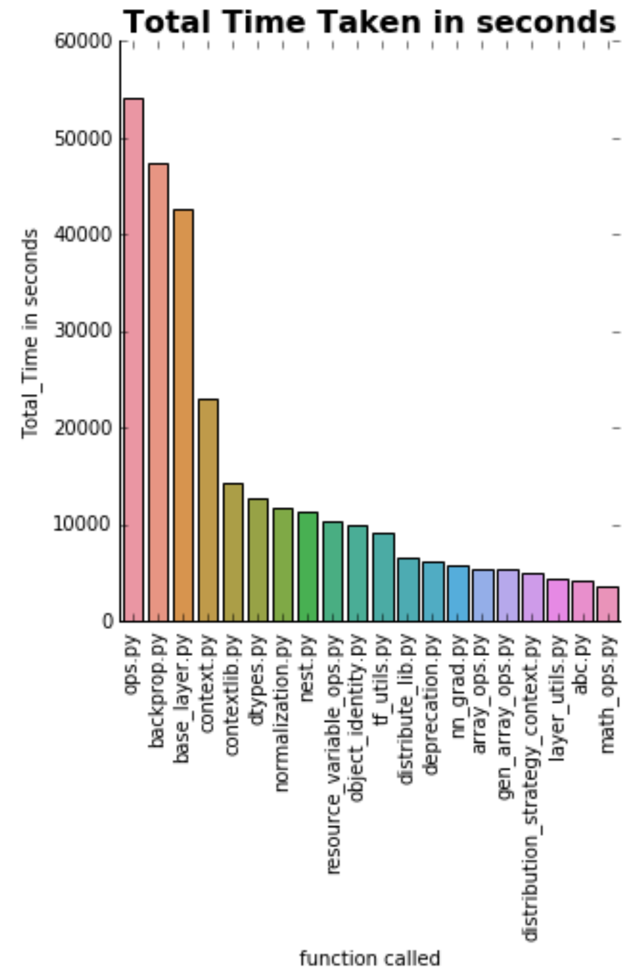




https://www.pluralsight.com/guides/time-profiling-neural-networks-model

Motivation





https://www.pluralsight.com/guides/time-profiling-neural-networks-model

Motivation

```cpp
#include <iostream>
#include <vector>
#include "mkldnn.hpp"

using namespace mkldnn;

…

int main(int argc, char **argv) {
    try {
        simple_net();
        std::cout << "ok\n";
    } catch (error &e) {
        std::cerr << "status: " << e.status << std::endl;
        std::cerr << "message: " << e.message << std::endl;
    }
    return 0;
}
```

https://oneapi-src.github.io/oneDNN/v1.0/cpu_performance_profiling_8cpp-example.html

Parallel Programming

For numerical linear algebra

In the second lecture, we have seen that within shared-memory parallel programming, we have broadly four options:

- **Confinement**: Do not share memory between threads.
- **Immutability**: Do not share any mutable data between threads.
- **Thread-safe code**: Use data types with additional guarantees for storing any mutable data shared between threads, or even better, use implementations of algorithms that are already parallelized and handle the concurrency issues for you.
- **Synchronization**: Use synchronization primitives to prevent accessing the variable at the same time.

BLAS

For prototyping numerical linear algebra

A key tool within linear algebra and machine learning are
two ancient specifications, known as:

BLAS (``Basic Linear Algebra Subprograms''),
which covers vector addition, dot products, and linear combinations (this dates back to 1979).
Level 2 added support for vector-matrix operations (1986),
and level 3 added support for matrix-matrix operations and
block-partitioned algorithms (1988).

LAPACK (``Linear Algebra Package''), which
covers matrix factorizations (LU, Cholesky and QR),
eigenvalue and least squares solvers.

BLAS

For prototyping numerical linear algebra

BLAS and LAPACK subroutines are all named naaop, where:

n suggests whether to use real floating-point numbers in
single (S) or double (D) precision,
or complex number with single (C)
or double (Z) precision.
aa denotes the assumptions on the matrix, e.g.,
diagonal (DI) specified by a vector, and
general matrix (GE).
op denotes the algorithm, e.g.,
matrix-matrix multiplication (MM), and
solving linear system (SV).

SGEMM is thus matrix-matrix multiplication of general dense matrices in single precision, and DDOT is vector-vector dot product in double precision.

BLAS

For prototyping numerical linear algebra

Level 1 BLAS

	dim	scalar	vector	vector	scalars	5-element array		prefixes
SUBROUTINE xROTG (					A, B, C, S)		Generate plane rotation	S, D
SUBROUTINE xROTMG(					D1, D2, A, B,	PARAM)	Generate modified plane rotation	S, D
SUBROUTINE xROT (	N,		X, INCX,	Y, INCY,	C, S)		Apply plane rotation	S, D
SUBROUTINE xROTM (	N,		X, INCX,	Y, INCY,		PARAM)	Apply modified plane rotation	S, D
SUBROUTINE xSWAP (	N,		X, INCX,	Y, INCY)			$x \leftrightarrow y$	S, D, C, Z
SUBROUTINE xSCAL (	N,	ALPHA,	X, INCX)				$x \leftarrow \alpha x$	S, D, C, Z, CS, ZD
SUBROUTINE xCOPY (	N,		X, INCX,	Y, INCY)			$y \leftarrow x$	S, D, C, Z
SUBROUTINE xAXPY (	N,	ALPHA,	X, INCX,	Y, INCY)			$y \leftarrow \alpha x + y$	S, D, C, Z
FUNCTION xDOT (	N,		X, INCX,	Y, INCY)			$dot \leftarrow x^T y$	S, D, DS
FUNCTION xDOTU (	N,		X, INCX,	Y, INCY)			$dot \leftarrow x^T y$	C, Z
FUNCTION xDOTC (	N,		X, INCX,	Y, INCY)			$dot \leftarrow x^H y$	C, Z
FUNCTION xxDOT (	N,		X, INCX,	Y, INCY)			$dot \leftarrow \alpha + x^T y$	SDS
FUNCTION xNRM2 (	N,		X, INCX)				$nrm2 \leftarrow \|\|x\|\|_2$	S, D, SC, DZ
FUNCTION xASUM (	N,		X, INCX)				$asum \leftarrow \|\|re(x)\|\|_1 + \|\|im(x)\|\|_1$	S, D, SC, DZ
FUNCTION IxAMAX(	N,		X, INCX)				$amax \leftarrow 1^{st}\, k \ni \|re(x_k)\| + \|im(x_k)\| = max(\|re(x_i)\| + \|im(x_i)\|)$	S, D, C, Z

Level 2 BLAS

	options	dim	b-width	scalar	matrix	vector	scalar	vector		
xGEMV (	TRANS,	M, N,		ALPHA,	A, LDA,	X, INCX,	BETA,	Y, INCY)	$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$	S, D, C, Z
xGBMV (	TRANS,	M, N,	KL, KU,	ALPHA,	A, LDA,	X, INCX,	BETA,	Y, INCY)	$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$	S, D, C, Z
xHEMV (	UPLO,	N,		ALPHA,	A, LDA,	X, INCX,	BETA,	Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	C, Z
xHBMV (	UPLO,	N,	K,	ALPHA,	A, LDA,	X, INCX,	BETA,	Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	C, Z
xHPMV (	UPLO,	N,		ALPHA,	AP,	X, INCX,	BETA,	Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	C, Z
xSYMV (	UPLO,	N,		ALPHA,	A, LDA,	X, INCX,	BETA,	Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	S, D
xSBMV (	UPLO,	N,	K,	ALPHA,	A, LDA,	X, INCX,	BETA,	Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	S, D
xSPMV (	UPLO,	N,		ALPHA,	AP,	X, INCX,	BETA,	Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	S, D
xTRMV (	UPLO, TRANS, DIAG,	N,			A, LDA,	X, INCX)			$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
xTBMV (	UPLO, TRANS, DIAG,	N,	K,		A, LDA,	X, INCX)			$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
xTPMV (	UPLO, TRANS, DIAG,	N,			AP,	X, INCX)			$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
xTRSV (	UPLO, TRANS, DIAG,	N,			A, LDA,	X, INCX)			$x \leftarrow A^{-1} x, x \leftarrow A^{-T} x, x \leftarrow A^{-H} x$	S, D, C, Z
xTBSV (	UPLO, TRANS, DIAG,	N,	K,		A, LDA,	X, INCX)			$x \leftarrow A^{-1} x, x \leftarrow A^{-T} x, x \leftarrow A^{-H} x$	S, D, C, Z
xTPSV (	UPLO, TRANS, DIAG,	N,			AP,	X, INCX)			$x \leftarrow A^{-1} x, x \leftarrow A^{-T} x, x \leftarrow A^{-H} x$	S, D, C, Z

	options	dim	scalar	vector	vector	matrix		
xGER (		M, N,	ALPHA,	X, INCX,	Y, INCY,	A, LDA)	$A \leftarrow \alpha xy^T + A, A - m \times n$	S, D
xGERU (		M, N,	ALPHA,	X, INCX,	Y, INCY,	A, LDA)	$A \leftarrow \alpha xy^T + A, A - m \times n$	C, Z
xGERC (		M, N,	ALPHA,	X, INCX,	Y, INCY,	A, LDA)	$A \leftarrow \alpha xy^H + A, A - m \times n$	C, Z
xHER (	UPLO,	N,	ALPHA,	X, INCX,		A, LDA)	$A \leftarrow \alpha xx^H + A$	C, Z
xHPR (	UPLO,	N,	ALPHA,	X, INCX,		AP)	$A \leftarrow \alpha xx^H + A$	C, Z
xHER2 (	UPLO,	N,	ALPHA,	X, INCX,	Y, INCY,	A, LDA)	$A \leftarrow \alpha xy^H + y(\alpha x)^H + A$	C, Z
xHPR2 (	UPLO,	N,	ALPHA,	X, INCX,	Y, INCY,	AP)	$A \leftarrow \alpha xy^H + y(\alpha x)^H + A$	C, Z
xSYR (	UPLO,	N,	ALPHA,	X, INCX,		A, LDA)	$A \leftarrow \alpha xx^T + A$	S, D
xSPR (	UPLO,	N,	ALPHA,	X, INCX,		AP)	$A \leftarrow \alpha xx^T + A$	S, D
xSYR2 (	UPLO,	N,	ALPHA,	X, INCX,	Y, INCY,	A, LDA)	$A \leftarrow \alpha xy^T + \alpha yx^T + A$	S, D
xSPR2 (	UPLO,	N,	ALPHA,	X, INCX,	Y, INCY,	AP)	$A \leftarrow \alpha xy^T + \alpha yx^T + A$	S, D

http://www.netlib.org/blas/blasqr.pdf

BLAS

For prototyping numerical linear algebra

Most vendors of acclerators maintain their own BLAS
- implementation:
- AMD maintains rocBLAS,
- Apple maintains Accelerate,
- ARM maintains Arm Performance Libraries,
- Intel develops Intel Math Kernel Library (iMKL), and
- NVIDIA maintains cuBLAS and NVBLAS.

Developer Reference for Intel® oneAPI Math Kernel Library - C

Developer Reference

Version: 2021.2 Last Updated: 03/26/2021 Public Content Download as PDF

Search this document

Developer Reference for Intel® oneAPI Math Kernel Library
Getting Help and Support
What's New
Notational Conventions
Overview
OpenMP* Offload
BLAS and Sparse BLAS Routines
BLAS Routines
Routine Naming Conventions

C Interface Conventions

cblas_?gemv

Computes a matrix-vector product using a general matrix.

Syntax

```
void cblas_sgemv (const CBLAS_LAYOUT Layoutconst CBLAS_TRANSPOSE transconst MKL_INT mconst MKL_INT nconst
float alphaconst float *aconst MKL_INT ldaconst float *xconst MKL_INT incxconst float betafloat *yconst
MKL_INT incy);

void cblas_dgemv (const CBLAS_LAYOUT Layoutconst CBLAS_TRANSPOSE transconst MKL_INT mconst MKL_INT nconst
double alphaconst double *aconst MKL_INT ldaconst double *xconst MKL_INT incxconst double betadouble
*yconst MKL_INT incy);

void cblas_cgemv (const CBLAS_LAYOUT Layoutconst CBLAS_TRANSPOSE transconst MKL_INT mconst MKL_INT nconst
void *alphaconst void *aconst MKL_INT ldaconst void *xconst MKL_INT incxconst void *betavoid *yconst
MKL_INT incy);
```

BLAS

For prototyping numerical linear algebra

Most vendors of acclerators maintain their own BLAS

- implementation:
- AMD maintains rocBLAS,
- Apple maintains Accelerate,
- ARM maintains Arm Performance Libraries,
- Intel develops Intel Math Kernel Library (iMKL), and
- NVIDIA maintains cuBLAS and NVBLAS.

Notable open-source implementation focussing mostly on CPUs are ATLAS, BLIS (BLAS-like Library Instantiation Software), and OpenBLAS. A special mention should be devoted to the ATLAS library, which automatically optimizes itself for any architecture, including complicated cache hierarchies.

BLAS

For prototyping numerical linear algebra

For C++ , there are multiple implementations of BLAS, loosely speaking.

Libraries such as Armadillo, eigen, Intel OneAPI, LAPACK++, uBlas are sometimes linked against ancient Fortran code, but provide decent C++ interfaces.

eigen and CLBlast actually provide C++ implementations too.

Intel OneAPI Mathematical Kernels comes with excellent documentation, examples, and support, but rather cumbersome naming conventions.

BLAS

For prototyping numerical linear algebra

```
1  #include <iostream>
2  #include <vector>
3  #include "mkldnn.hpp"
4
5  using namespace mkldnn;
6  using dim_t = mkldnn::memory::dim;
7
8  int main(int argc, char **argv) {
9      try {
10         const dim_t n = 64;
11         // column-major order (sloupce souvisle)
12         std::vector<float> A(n*n, 1.0f);
13         std::vector<float> B(n*n, 1.0f);
14         std::vector<float> C(n*n, 1.0f);
```

BLAS

For prototyping numerical linear algebra

```
15          //
            ↪  https://oneapi-src.github.io/oneDNN/v0/group__c__api_
16          mkldnn_status_t status = mkldnn_sgemm('N', 'N', n, n,
            ↪  n,
17                          1.f, A.data(), n, B.data(), n,
18                          0.f, C.data(), n);
19          std::cerr << "status: " << status << std::endl;
20      } catch (error &e) {
21          std::cerr << "status: " << e.status << std::endl;
22          std::cerr << "message: " << e.message << std::endl;
23      }
24      return 0;
25  }
26
```

In contrast, Boost.org uBlas provides a much more modern C++20-only syntax, and can be linked to arbitrary vendor-provided BLAS library.

BLAS

For prototyping numerical linear algebra

```cpp
1  #include <boost/numeric/ublas/tensor.hpp>
2  #include <iostream>
3
4  int main()
5  {
6    using namespace boost::numeric::ublas::index;
7    using tensor  =
   ↪  boost::numeric::ublas::tensor_dynamic<float>;
8    auto ones     = boost::numeric::ublas::ones<float>{};
9
10   tensor A = ones(64,64);
11   tensor B = ones(64,64);
12
13   tensor C = A(_i,_j)*B(_i,_j);
14   std::cout << "C=" << C << ";" << std::endl;
15 }
```

BLAS

For prototyping numerical linear algebra

```
1  #include <boost/numeric/ublas/tensor.hpp>
2  #include <iostream>
3
4  int main()
5  {
6    using namespace boost::numeric::ublas::index;
7    using tensor  =
   ↪  boost::numeric::ublas::tensor_dynamic<float>;
8    auto ones     = boost::numeric::ublas::ones<float>{};
9
10   tensor A = ones(3,4,5);
11   tensor B = ones(4,6,3,2);
12
13   tensor C = 2*ones(5,6,2) + A(_i,_j,_k)*B(_j,_l,_i,_m) + 5;
14   std::cout << "C=" << C << ";" << std::endl;
15 }
```

BLAS

For prototyping numerical linear algebra

As you may know, the tensor computations underlie much of modern machine learning, in the form of training deep neural networks. The TensorFlow and PyTorch are key contenders there, again easy to link against any vendor-provided BLAS.

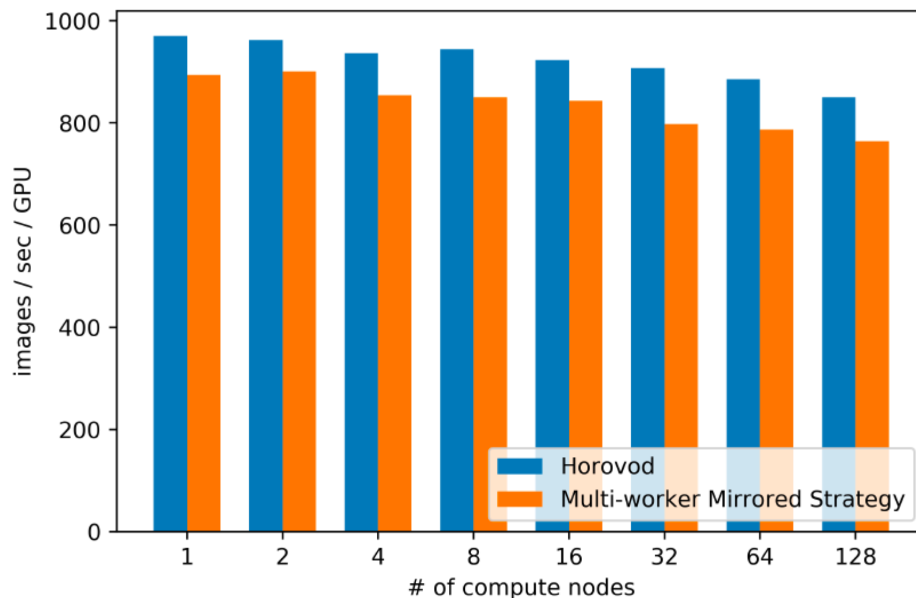
Libraries such as TensorFlow make make it possible to exploit much of the theoretically available processing power. See, for example, Summit, a supercomputer at the Oak Ridge National Laboratory in Tennessee, USA.
It 9,216 POWER9 22-core CPUs and 27,648 NVIDIA Tesla V100 GPUs, each of which has 5,120 CUDA Cores. In total, this means 141+ million cores with circa 200 petaFLOPS performance.

BLAS

For prototyping numerical linear algebra

As you may know, the tensor computations underlie much of modern machine learning, in the form of training deep neural networks. The TensorFlow and PyTorch are key contenders there, again easy to link against any vendor-provided BLAS.

Libraries such as TensorFlow make make it possible to exploit much of the theoretically available processing power. Consider the scaling for training ResNet deep neural network on ImageNet benchmark.



D Distributed Deep Learning Examples
Project ID: 5605
Star 1
29 Commits 1 Branch 0 Tags 22.7 MB Project Storage
distributed deep learning examples on Summit for Keras, PyTorch, and TensorFlow.
add vis notebook
Yin, Junqi authored 1 year ago
b1f1f2df
master distributed-deep-learning-examples
Find file Clone
README

https://code.ornl.gov/olcf-analytics/summit/distributed-deep-learning-examples

Parallel Programming

For numerical linear algebra

In the second lecture, we have seen that within shared-memory parallel programming, we have broadly four options:

- **Confinement**: Do not share memory between threads.
- **Immutability**: Do not share any mutable data between threads.
- **Thread-safe code**: Use data types with additional guarantees for storing any mutable data shared between threads, or even better, use implementations of algorithms that are already parallelized and handle the concurrency issues for you.
- **Synchronization**: Use synchronization primitives to prevent accessing the variable at the same time.

Matrix-vector Multiplication

The easy part

a11	a12	a13	a14	a15		x1		y1
a21	…					x2		y2
…					X	x3	=	y3
						x4		y4
				a55		x5		y5

$$y_1 = \sum_{\{i=1,\ldots,5\}} a_{1i} \, . \, x_i$$

Matrix-vector Multiplication

The easy part

Embarrassingly parallel:

P1 →	a11	a12	a13	a14	a15		x1		y1
P2 →	a21	…					x2		y2
	…					X	x3	=	y3
…							x4		y4
					a55		x5		y5

$$y_1 = \sum_{\{i=1,\dots,5\}} a_{1i} \, . \, x_i$$

Matrix-vector Multiplication

The easy part

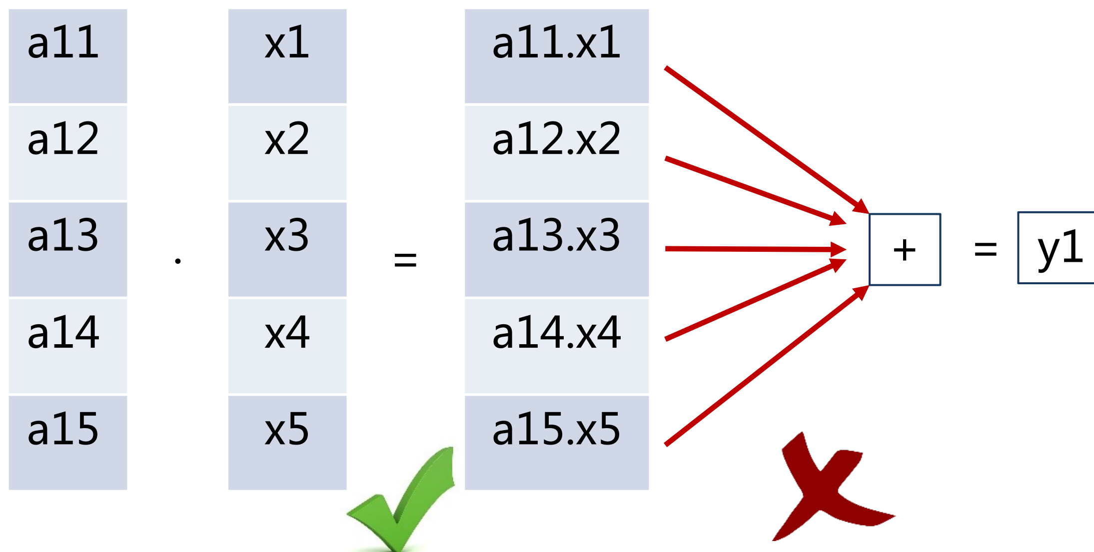
Can you vectorize this?

```
1  void multiply(std::vector<int>& A, std::vector<int>& x,
   ↪  std::vector<int>& y) {
2  #pragma omp declare reduction(vec_int_plus : std::vector<int>
   ↪  : std::transform(omp_out.begin(), omp_out.end(),
   ↪  omp_in.begin(), omp_out.begin(), std::plus<int>()))
   ↪  initializer(omp_priv = omp_orig)
3
4  #pragma omp parallel for num_threads(thread_count)
   ↪  reduction(vec_int_plus : y)
5      for (int i=0; i<ROWS; i++) {
6          for (int j=0; j<COLS; j++) {
7              y[i] += A[i * COLS + j]*x[j];
8          }
9      }
10 }
```

Matrix-vector Multiplication

The easy part

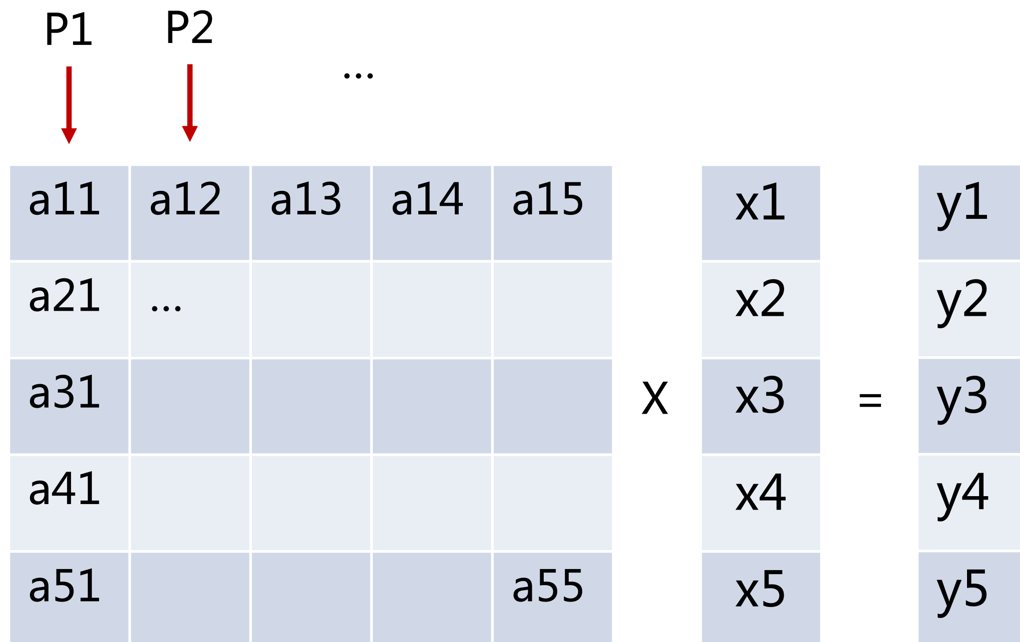
```
4  #pragma omp parallel for num_threads(thread_count)
   ↪   reduction(vec_int_plus : y)
5      for (int i=0; i<ROWS; i++) {
6          for (int j=0; j<COLS; j++) {
7              y[i] += A[i * COLS + j]*x[j];
8          }
9      }
10 }
```



Matrix-vector Multiplication

The not-so-easy part

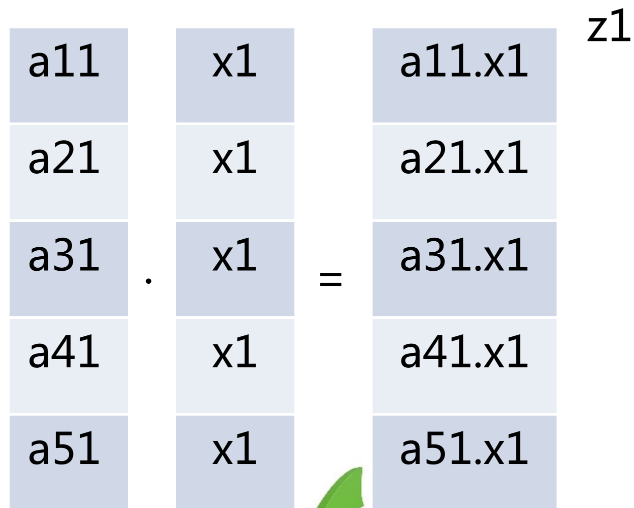
- We could pre-compute helper variables column-wise:

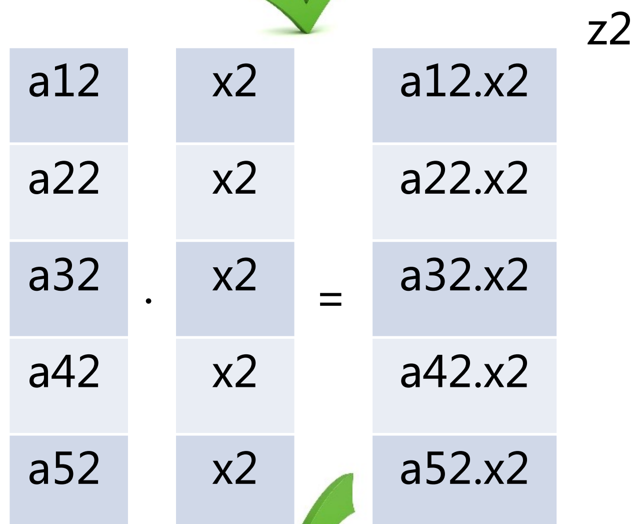


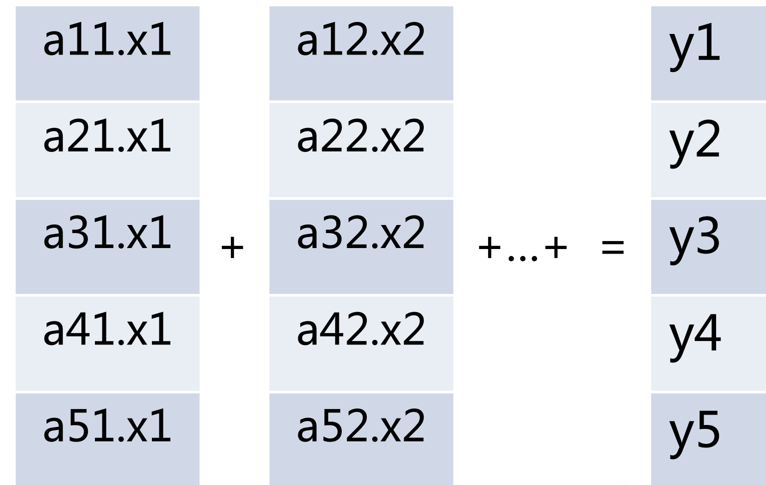
$$z_{ij} = a_{ij} . x_j \qquad y_i = \sum_{\{j=1,\ldots,5\}} z_{ij}$$

Matrix-vector Multiplication

The not-so-easy part







What is not to like?

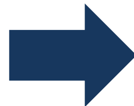
Matrix-vector Multiplication

The not-so-easy part

- We can reorder the matrices

```
1  ...
2  // data has to be ordered by columns in memory
3  for (int i = 0; i < COLS; i++) {
4              x[i] = rand() % 1000;
5              for (int j = 0; j < ROWS; j++) {
6                  A[i * ROWS + j] = rand() % 1000;
7              }
8          }
9  ...
10
```

a11	a12	a13	a14	a15
a21	...			
a31				
a41				
a51				a55



a11	a21	a31	a41	a51	...		

Matrix-vector Multiplication

The not-so-easy part

- We can reorder the matrices

```
14  #pragma omp parallel for num_threads(thread_count)
    ↪   reduction(vec_int_plus : y)
15      for ( int i = 0; i < COLS ; i ++) {
16          for (int j = 0; j < ROWS; j++) {
17              y[j] += A[i * ROWS + j]*x[i];
18          }
19      }
20  }
```

Matrix-vector Multiplication

Trivial tricks are best

```
1  void multiply(std::vector<int> &A, std::vector<int> &x,
   ↪  std::vector<int> &y) {
2  #pragma omp declare reduction(vec_int_plus : std::vector<int>
   ↪  : std::transform(omp_out.begin(), omp_out.end(),
   ↪  omp_in.begin(), omp_out.begin(), std::plus<int>()))
   ↪  initializer(omp_priv = omp_orig)
3
4      int tmp;
5  #pragma omp parallel for num_threads(thread_count)
   ↪  reduction(vec_int_plus : y)
6      for (int i=0; i<ROWS; i++) {
7          tmp = 0;
8          for (int j=0; j<COLS; j++) {
9              tmp += A[i * COLS + j]*x[j];
10         }
11         y[i] += tmp;
12     }
13 }
```

Matrix-vector Multiplication

The Actual Implementations

Computer Languages, Systems & Structures 51 (2018) 158–175

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Effective Implementation of Matrix–Vector Multiplication on Intel's AVX multicore Processor

Somaia A. Hassan[a,*], Mountasser M.M. Mahmoud[a], A.M. Hemeida[b], Mahmoud A. Saber[a]

Matrix-Matrix Multiplication

The definition

a11	a12	a13
a21	…	
…		

X

b11	b12	b13
b21	…	
…		

=

c11	c12	c13
c21	…	
…		

Also embarrassingly parallel:

$$c_{ij} = \sum_{\{k=1,\dots,n\}} a_{ik} \,.\, b_{kj}$$

but scheduling is a lot harder.

Matrix-Matrix Multiplication

The first parallel version

```
1  void multiply(std::vector<int>& A, std::vector<int>& B,
   ↪  std::vector<int>& C) {
2  #pragma omp declare reduction(vec_int_plus : std::vector<int>
   ↪  : std::transform(omp_out.begin(), omp_out.end(),
   ↪  omp_in.begin(), omp_out.begin(), std::plus<int>()))
   ↪  initializer(omp_priv = omp_orig)
3      int tmp;
4  #pragma omp parallel for collapse(2) num_threads(thread_count)
   ↪  reduction(vec_int_plus : C)
5      for (int i=0; i<ROWS; i++) {
6          for (int j=0; j<COLS; j++) {
7              tmp = 0;
8              for (int k=0; k<ROWS; k++) {
9                  tmp += A[i * COLS + k] * B[k * COLS + j];
10             }
11             C[i * COLS + j] += tmp;
12         }
13     }
14 }
```

Matrix-Matrix Multiplication

The second parallel version

- Division into blocks (submatrices)

b11	b12	b13	b14
b21	…		
…			

a11	a12	a13	a14
a21	…		
…			

c11	c12	c13	c14
c21	…		
…			

$$c_{11} += a_{11}.b_{11} + a_{12}.b_{21}$$
$$c_{12} += a_{11}.b_{12} + a_{12}.b_{22}$$
…

Matrix-Matrix Multiplication

The second parallel version

```
1  void multiply_blocks(std::vector<int>& A, std::vector<int>& B,
   ↪  std::vector<int>& C) {
2  #pragma omp declare reduction(vec_int_plus : std::vector<int>
   ↪  : std::transform(omp_out.begin(), omp_out.end(),
   ↪  omp_in.begin(), omp_out.begin(), std::plus<int>()))
   ↪  initializer(omp_priv = omp_orig)
3
4      const int ROWS_IN_BLOCK = 10;
5      const int BLOCKS_IN_ROW = ROWS/ROWS_IN_BLOCK;
6      int tmp;
7
8  #pragma omp parallel for collapse(2) num_threads(thread_count)
   ↪  reduction(vec_int_plus : C) private(tmp)
9      for (int br1=0; br1<BLOCKS_IN_ROW; br1++) {
10         for (int bb=0; bb<BLOCKS_IN_ROW; bb++) {
11             for (int bc2 = 0; bc2 < BLOCKS_IN_ROW; bc2++) {
12                 for (int r = br1 * ROWS_IN_BLOCK; r < (br1 +
                   ↪  1) * ROWS_IN_BLOCK; r++) {
13                     for (int c = bc2 * ROWS_IN_BLOCK; c < (bc2
                       ↪  + 1) * ROWS_IN_BLOCK; c++) {
14                         tmp = 0;
```

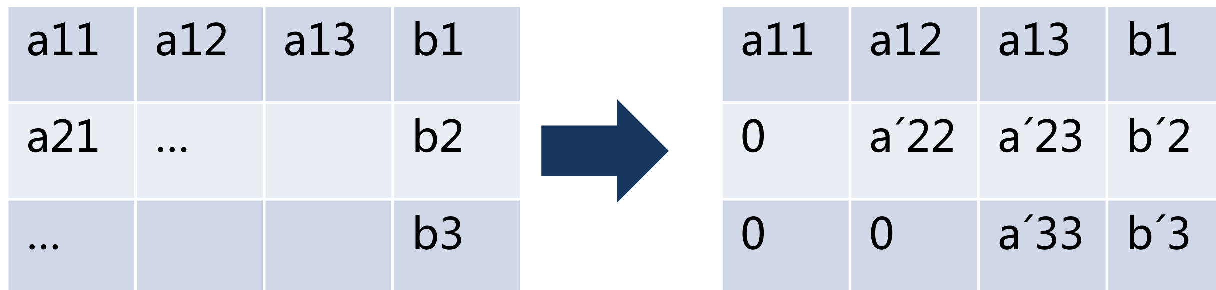
Matrix-Matrix Multiplication

The second parallel version

```
15                                  for (int k = 0; k < ROWS_IN_BLOCK;
                                    ↪   k++) {
16                                      tmp += A[r * COLS + (k +
                                        ↪   bb*ROWS_IN_BLOCK)] *
                                        ↪   B[(bb*ROWS_IN_BLOCK + k) *
                                        ↪   COLS + c];
17                                  }
18                                  C[r * COLS + c] += tmp;
19                              }
20                          }
21                      }
22                  }
23              }
24      }
```

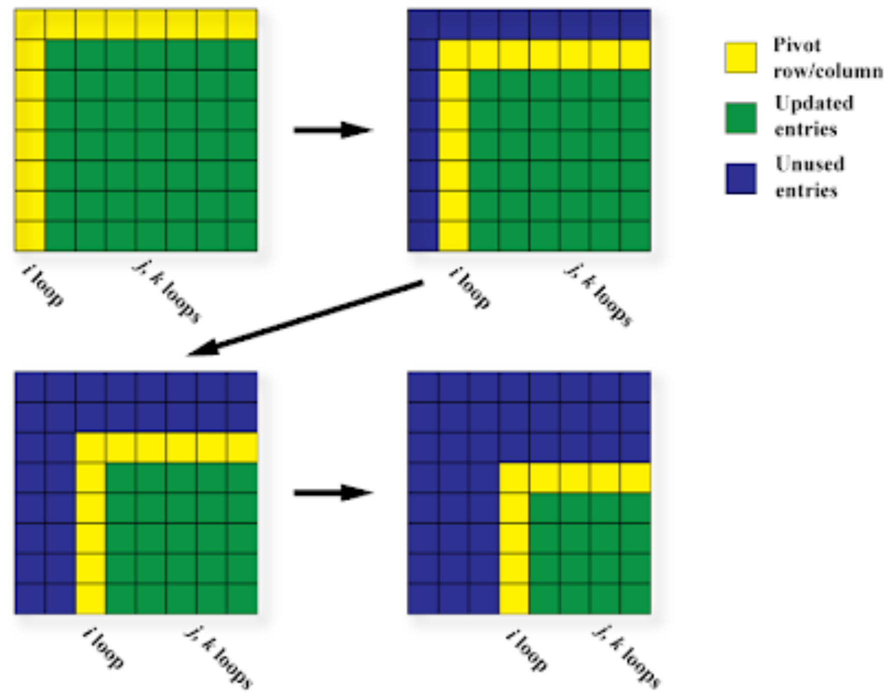
Solving Linear Systems

Gauss elimination



a11	a12	a13	b1
a21	…		b2
…			b3

a11	a12	a13	b1
0	a´22	a´23	b´2
0	0	a´33	b´3



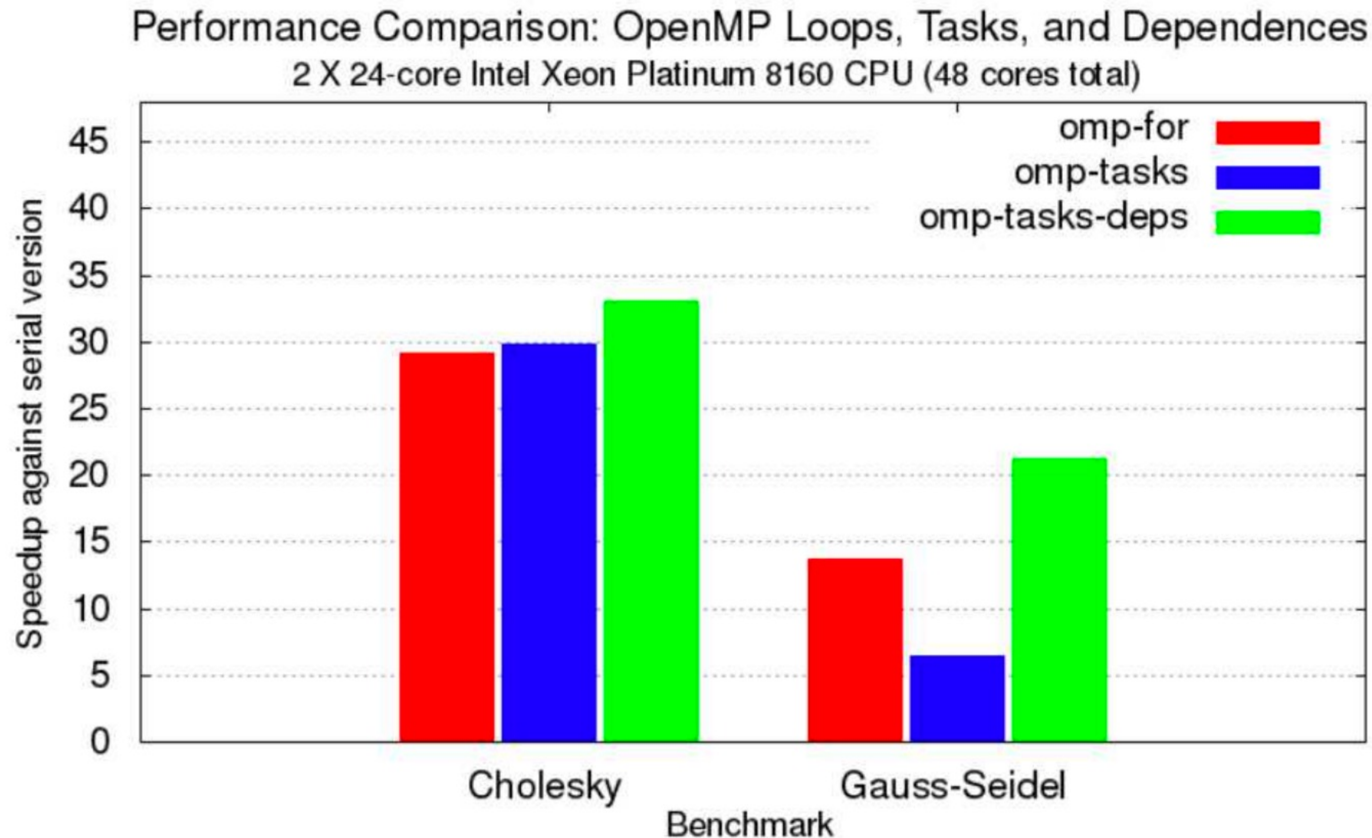
Solving Linear Systems

Gauss elimination

```
1  void gauss_par(std::vector<double>& A) {
2  #pragma omp declare reduction(vec_int_plus :
   ↪  std::vector<double> : std::transform(omp_out.begin(),
   ↪  omp_out.end(), omp_in.begin(), omp_out.begin(),
   ↪  std::plus<double>())) initializer(omp_priv = omp_orig)
3      for (int i=0; i<ROWS; i++) {
4          // Make all rows below this one 0 in current column
5  #pragma omp parallel for num_threads(thread_count)
6          for (int k=i+1; k<ROWS; k++) {
7              double c = -A[k * COLS + i]/A[i*COLS + i];
8              for (int j=i; j<ROWS; j++) {
9                  if (i==j) {
10                     A[k * COLS + j] = 0;
11                 } else {
12                     A[k * COLS + j] += c * A[i * COLS + j];
13                 }
14             }
15         }
16     }
17 }
```

Solving Linear Systems

Remember the Ongoing Evlution of OpenMP?



"The Ongoing Evolution of OpenMP"
https://www.osti.gov/pages/servlets/purl/1465188

Solving Linear Systems

Via Solving Least Squares

SIAM J. Matrix Anal. Appl.
Vol. 36, No. 4, pp. 1660–1690

© 2015 Robert M. Gower and Peter Richtárik. Published by SIAM
under the terms of the Creative Commons 4.0 license

RANDOMIZED ITERATIVE METHODS FOR LINEAR SYSTEMS*

ROBERT M. GOWER† AND PETER RICHTÁRIK†

Abstract. We develop a novel, fundamental, and surprisingly simple *randomized iterative method* for solving consistent linear systems. Our method has six different but equivalent interpretations: sketch-and-project, constrain-and-approximate, random intersect, random linear solve, random update, and random fixed point. By varying its two parameters—a positive definite matrix (defining geometry), and a random matrix (sampled in an independent and identically distributed fashion in each iteration)—we recover a comprehensive array of well-known algorithms as special cases, including the randomized Kaczmarz method, randomized Newton method, randomized coordinate descent method, and random Gaussian pursuit. We naturally also obtain variants of all these methods using blocks and importance sampling. However, our method allows for a much wider selection of these two parameters, which leads to a number of new specific methods. We prove exponential convergence of the *expected norm of the error* in a single theorem, from which existing complexity results for known variants can be obtained. However, we also give an exact formula for the evolution of the expected iterates, which allows us to give *lower bounds* on the convergence rate.

Solving Linear Systems

Via Solving Least Squares

$$\mathbf{E}\left[x^{k+1} - x^*\right] = \left(I - B^{-1}\mathbf{E}\left[Z\right]\right)\mathbf{E}\left[x^k - x^*\right]$$

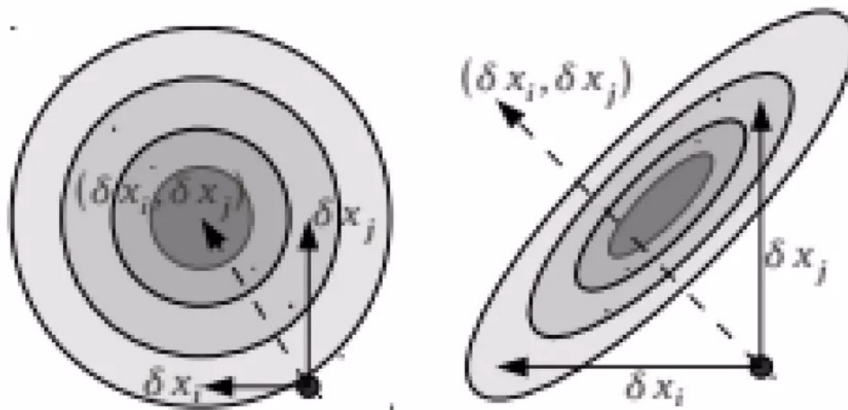
$$\|\mathbf{E}\left[x^{k+1} - x^*\right]\|_B \leq \rho \cdot \|\mathbf{E}\left[x^k - x^*\right]\|_B$$

$$\mathbf{E}\left[\|x^{k+1} - x^*\|_B^2\right] \leq \rho \cdot \mathbf{E}\left[\|x^k - x^*\|_B^2\right]$$

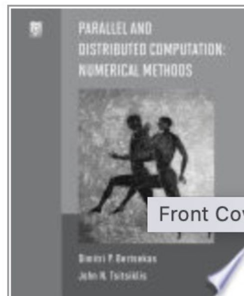
$$\rho = 1 - \lambda_{\min}(B^{-1/2}\mathbf{E}\left[Z\right]B^{-1/2}).$$

Optimization

Parallel Aspects



Parallel and Distributed Computation: Numerical Methods



Dimitri Bertsekas, John Tsitsiklis

Athena Scientific, Mar 1, 2015 - Mathematics - 735 pages

★★★★★

0 Reviews ⓘ

This highly acclaimed work, first published by Prentice Hall in 1989, is a comprehensive and theore ethods. It focuses on algorithms that are naturally suited for massive parallelization, and it explore communication, and synchronization issues associated with such algorithms.

https://pt.slideshare.net/ShaleenGupta6/parallel-coordinate-descent-algorithms

Next Steps

Parallel Programming (B4M35PAG)
GPGPUs (B4M39GPU)
Bc Dissertation
State Exams
MSc Dissertation
State Exams
...
PhD
...
Real-world?

Next Steps

State Exams

Hardwarová podpora pro paralelní výpočty: (super)skalární architektury, pipelining, spekulativní vyhodnocování, vektorové instrukce, vlákna, procesy, GPGPU. Hierarchie cache pamětí.

Komplikace v paralelním programování: souběh (race condition), uváznutí (deadlock), iluze sdílení (false sharing).

Podpora paralelního programování v C a C++: pthreads, thread, jthread, atomic, mutex, lock_guard.

Podpora paralelního programování v OpenMP: sériově-paralelní model uspořádání vláken (fork-join), paralelizovatelná úloha (task region), různé implementace specifikace. Direktivy parallel, for, section, task, barrier, critical, atomic.

Techniky dekompozice programu: statické a paralelní rozdělení práce. Threadpool a fronta úkolů. Balancování a závislosti (dependencies).

Techniky dekompozice programu na příkladech z řazení: quick sort, merge sort.

Techniky dekompozice programu na příkladech z numerické lineární algebry a strojového učení: násobení matice vektorem, násobení dvou matic, řešení systému lineárních rovnic.