Concepts and Models of Parallel and Data-centric Programming

Parallel Algorithms II

Lecture, Summer 2020

Dr. Christian Terboven <terboven@itc.rwth-aachen.de>

i12 High Performance Computing | RWTH AACHEN UNIVERSITY

Outline

0. Organization
1. Foundations
2. Shared Memory
3. GPU Programming
4. Bulk-Synchronous Parallelism
5. Message Passing
6. Distributed Shared Memory
7. **Parallel Algorithms**
8. Parallel I/O
9. MapReduce
10. Apache Spark

a. Berkeley DWARFS
b. Dense Linear Algebra
c. Sparse Linear Algebra
d. Monte Carlo Methods
e. Graph Traversal

Dense Linear Algebra

Overview

- Linear algebra is often a fundamental part in engineering and computer science

- Basic problems
  - Linear systems: $\boldsymbol{Ax} = \boldsymbol{b}$
  - Least squares: minimize $\|\boldsymbol{Ax} - \boldsymbol{b}\|_2$
  - Eigenvalues: $\boldsymbol{Ax} = \boldsymbol{\lambda x}$
  - Singular values and vectors

- Definition: A dense matrix is neither sparse nor structured.

- Common linear algebra operations are specified in the Basic Linear Algebra Subprograms (BLAS)

History

- BLAS 1 (1973-1977)
  - Operate on vectors or pairs of vectors
  - O(n) operations on O(n) data
  - E.g. AXPY ($y = ax + y$), dot product, scale, vector norms
- BLAS 2 (1984-1986)
  - Operate on matrix/ vector pairs
  - $O(n^2)$ operations on $O(n^2)$ data
  - E.g. matrix-vector product
- BLAS 3 (1987-1988)
  - Operate on matrix/ matrix pairs
  - $O(n^3)$ operations on $O(n^2)$ data
  - E.g. matrix-matrix product

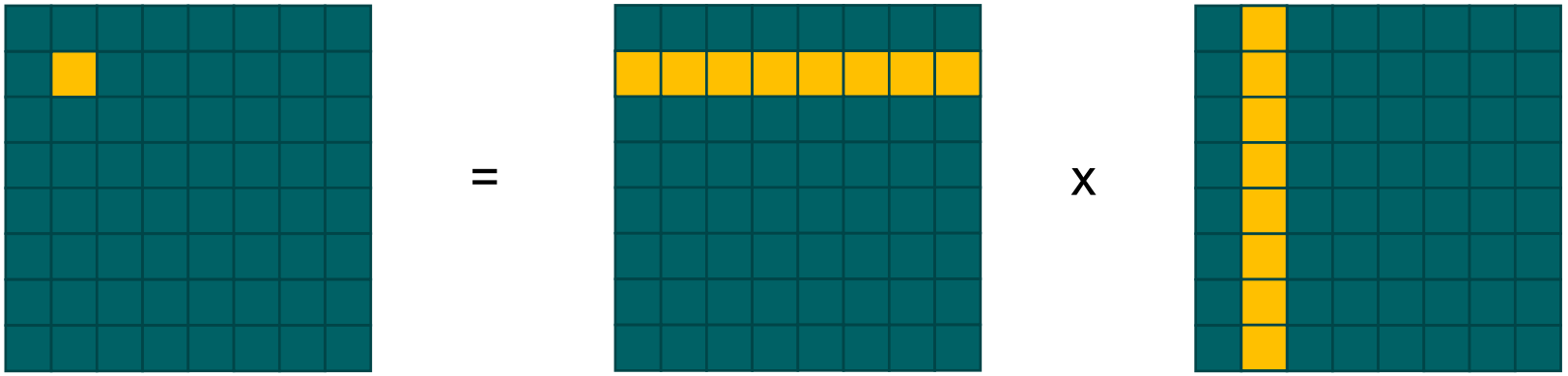
BLAS Papers:

- BLAS 1: C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, Basic Linear Algebra Subprograms for Fortran Usage, ACM Transactions on Mathematical Software, 5:308--325, 1979.
- J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson, An Extended Set of Fortran Basic Linear Algebra Subprograms, ACM Transactions on Mathematical Software, 14(1):1--32, 1988.
- J. Dongarra, J. Du Croz, I. Duff, S. Hammarling, A Set of Level 3 Basic Linear Algebra Subprograms, ACM Transactions on Mathematical Software, 16(1):1--17, 1990.

GEMM: Sequential Implementation

- GEMM: General Matrix-Matrix Multiplication (BLAS Level 3)
- Let $\boldsymbol{A} = [a_{ij}]_{n \times n}$ and $\boldsymbol{B} = [b_{ij}]_{n \times n}$ be $n \times n$ matrices. Compute $\boldsymbol{C} = \boldsymbol{AB}$

```
// Serial gemm
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    C[i][j] = 0.0;
    for (int k = 0; k < n; k++) {
      C[i][j] = C[i][j] + A[i][k]*B[k][j];
} } }
```



GEMM: Sequential Implementation

- Let $\boldsymbol{A} = [a_{ij}]_{n \times n}$ **and** $\boldsymbol{B} = [b_{ij}]_{n \times n}$ be $n \times n$ matrices. Compute $\boldsymbol{C} = \boldsymbol{AB}$

```
// Serial gemm
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    C[i][j] = 0.0;
    for (int k = 0; k < n; k++) {
      C[i][j] = C[i][j] + A[i][k]*B[k][j];
} } }
```

- Computational complexity of sequential algorithm: $O(n^3)$
- Inner loop: 2 FLOPs and 3 LOADs `C[i][j], A[i][k], B[k][j]`
- Kernel bound by global memory bandwidth

- How to optimize this code?

GEMM: Serial Optimizations

- Can this code be vectorized?
  - No data or computational dependencies
  - Regular data accesses

- How to reduce the number of LOADs from main memory?
  - A, B are accessed n times
  - Keep as much data as possible in caches
  - Cache blocking

GEMM: Cache Blocking

- Partitioning of A,B,C into blocks of size $bsize \times bsize$ where

$$bsize = \frac{n}{nblocks}$$

```
// Blocked gemm
for (int ib = 0; ib < n; ib += bsize) {
  for (int jb = 0; jb < n; jb += bsize) {
    for (int kb = 0; kb < n; kb += bsize) {
      C[i][j] = 0.0;
      for (int i = ib; i < min(ib + bsize, n); i++) {
        for (int k = kb; k < min(kb + bsize, n); k++) {
          for (int j = jb; j < min(jb + bsize, n); j++) {
            C[i][j] = C[i][j] + A[i][k]*B[k][j];
} } } } } }
```

- ib,jb,kb loops over blocks, optimization for cache reuse
- i,j,k loop over elements of submatrices, optimization for fast execution

GEMM: Shared Memory

- Embarrassingly parallel algorithm
  - No data or computational dependencies
  - Regular data accesses
  - Balanced work load

- Exemplary parallelization with OpenMP

```
// OpenMP gemm
#pragma omp parallel for
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    C[i][j] = 0.0;
    for (int k = 0; k < n; k++) {
      C[i][j] = C[i][j] + A[i][k]*B[k][j];
} } }
```

GEMM: GPU

- Simple offloading implementation
  1. Allocate memory for A,B,C on device
  2. Copy A,B,C from host to device
  3. Call kernel on device

```
__global__ void sgemm(float **A, float **B, float **C, uint n) {
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  if (row < n && col  < n) {
    C[row][col] = 0.0;
    for (int i = 0; i < n; ++i) {
      C[row][col] += A[row][i] * B[i][col];
} } }
```

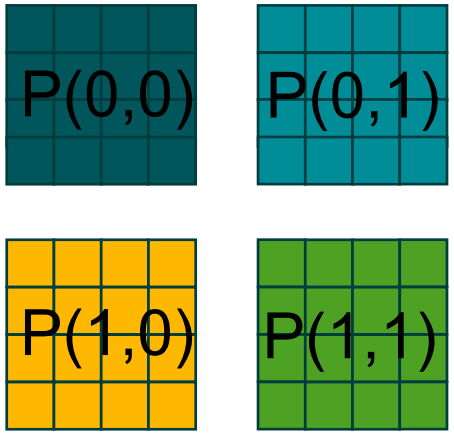
  4. Copy back result C to host
  5. Free memory of A,B,C on device

GEMM: Distributed Memory

- Partitioning of A, B in P (no. processors) square blocks of size $\frac{n}{\sqrt{P}} \times \frac{n}{\sqrt{P}}$
  - Alternative: row-wise or column-wise partitioning

- Each process $P_{i,j}$ performs $C_{i,j} = \sum_{k=0}^{\sqrt{P}} A_{i,k} B_{k,j}$ locally

- Exchange data A,B as needed with neighbors
  - All-to-all broadcast of blocks A in each row and blocks B in each column of processes
  - More advanced algorithms include Cannon, Fox, etc.



- Computational complexity: $O\left(\frac{n^3}{\sqrt{P}}\right)$

GEMM: Results

- Sample implementation with CUBLAS

```
$CUDA_PATH/samples/0_Simple/matrixMulCUBLAS/matrixMulCUBLAS
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla P100-SXM2-16GB" with compute capability 6.0

GPU Device 0: "Tesla P100-SXM2-16GB" with compute capability 6.0

MatrixA(1280,960), MatrixB(960,640), MatrixC(1280,640)
Computing result using CUBLAS...done.
Performance= 6849.72 GFlop/s, Time= 0.230 msec, Size= 1572864000
Ops
Computing result using host CPU...done.
Comparing CUBLAS Matrix Multiply with CPU results: PASS

NOTE: The CUDA Samples are not meant for performance measurements.
Results may vary when GPU Boost is enabled.
```

GEMM: Results

- Sample implementation with CUBLAS ~7 TFlop/s (SP)

- Peak performance of P100 ~9.5 TFlop/s (SP)
  - Close to optimal performance
  - Problem is well-suited for GPUs and throughput-oriented hardware in general

Summary Dense Linear Algebra

- Common in many algorithms and benchmarks (High-Performance Linpack)
- Fast implementation of most operations available
  - Libraries include BLAS, LAPACK, ScaLAPACK, SuperLU, ATLAS, Intel MKL, Eigen, cuBLAS
- Suitable for most architectures
  - Highest performance typically on throughput-oriented hardware