How To Optimize GEMM

The Ultimate Edition

Including analysis of ALL project files

Table of Contents

Chapter 1: Overview

File: README.md

Technical Analysis of "How To Optimize GEMM" README

1. Role & Purpose

This `README.md` file serves as the **central documentation hub** and **navigation portal** for a comprehensive educational resource on optimizing the General Matrix-Matrix Multiplication (GEMM) operation. Its primary functions within the project infrastructure are:

- **Project Index & Roadmap**: It provides a structured table of contents that outlines a complete **step-by-step optimization methodology**, guiding the reader from naive implementations to highly optimized ones.
- **Knowledge Repository Gateway**: It acts as the entry point to a GitHub Wiki, which contains the detailed technical content. The wiki format is ideal for this purpose as it supports **versioned, collaborative documentation** with rich formatting.
- **Academic Context & Attribution**: It clearly states the project's origins (Prof. Robert van de Geijn's group at UT Austin) and acknowledges funding sources (NSF grants), which is critical for **academic transparency and reproducibility**.

- **Resource Aggregator**: It links to related projects like **BLISlab** (a hands-on framework for GEMM optimization) and external tutorials, positioning itself within a broader ecosystem of HPC educational tools.

2. Technical Details

The logic and progression outlined in the table of contents reveal a layered optimization strategy, mirroring the **GotoBLAS/BLIS** philosophy, which is the de facto standard for high-performance dense linear algebra libraries.

- **Foundational Concept: Blocking for the Memory Hierarchy** The core logic is based on the principle that to achieve high performance, computation must be structured to respect the **memory hierarchy** (registers, L1/L2/L3 cache, main memory). The steps reflect this:
  1. **Micro-kernel Development** (`Computing four elements of C at a time`): This is the **innermost loop optimization**. The goal is to maximize **arithmetic intensity** (FLOPs/byte) within the CPU registers and L1 cache. Techniques here involve:
     - **Loop Unrolling**: Manually expanding loops to reduce branch overhead and create larger basic blocks for the compiler.
     - **Scalar Replacement**: Using local variables (which the compiler will place in registers) for accumulators instead of repeatedly dereferencing pointers to the C matrix in memory.
     - **Instruction-Level Parallelism (ILP)**: Scheduling multiple independent floating-point operations to keep the CPU's functional units busy.
  2. **Macro-kernel & Packing** (`Computing a 4x4 block of C at a time`, `Packing into contiguous memory`): This layer optimizes for the **L2/L3 cache**.
     - **Blocking (or Tiling)**: The matrices A, B, and C are conceptually divided into smaller blocks that fit into higher-level caches. The multiplication is then performed as a series of block-matrix multiplications.
     - **Packing**: This is a critical, non-intuitive optimization. Before computation, sub-blocks of matrix A (and often B) are copied into a **contiguous, cache-aligned temporary buffer**. This transformation:
       * Ensures the data accessed by the micro-kernel is in a **dense, unit-stride** pattern, maximizing cache line utilization and prefetcher efficiency.
       * Converts problems with non-standard leading dimensions (`lda`, `ldb`) into a uniform, optimized format.
       * **Amortizes the cost of TLB misses and cache conflict misses** over many more FLOPs.
- **Implementation Mechanics (Inferred from Titles)**:
  - **C Pointers**: The initial implementations likely use multi-

dimensional pointer arithmetic (`A + i*lda + k`) for row-major or column-major access. Optimization involves converting these to **single-strided accesses** within packed buffers.
  - **Subroutine Abstraction** (`Hiding computation in a subroutine`): A key software engineering practice. The highly optimized micro-kernel is isolated into a small, reusable function (often written in assembly or with intrinsics). This clean interface allows the macro-kernel to remain portable and readable.
  - **SIMD Intrinsics (AVX/SSE)**: While not explicitly stated in the TOC, the step `Further optimizing` and the linked external resource "GEMM: From Pure C to SSE Optimized Micro Kernels" strongly imply that the final micro-kernels use **SSE or AVX intrinsics** (e.g., `__m256d` types with `_mm256_fmadd_pd`) to exploit **data-level parallelism**. This allows performing 4 (AVX) or 8 (AVX-512) double-precision operations in a single instruction.

3. HPC Context

This specific document and the tutorial it organizes are profoundly relevant to high-performance computing for several reasons:

- **GEMM as a Computational Keystone**: The **DGEMM/SGEMM** (Double/Single-precision GEMM) routine is the **workhorse** of scientific computing, forming the core of LAPACK, machine learning (deep learning layers), and countless other applications. Its performance often dictates overall application speed.
- **Demonstration of Foundational Optimization Principles**: The step-by-step guide is not just about GEMM; it's a masterclass in general HPC optimization:
  - **Memory-Bound vs. Compute-Bound**: Naive GEMM is severely **memory-bound**, as it performs $O(n^3)$ operations on $O(n^2)$ data. The tutorial teaches how to transform it into a **compute-bound** problem within the cache hierarchy.
  - **Layered Optimization Strategy**: It exemplifies the "**meet-in-the-middle**" approach used in state-of-the-art libraries: a hand-tuned, architecture-specific micro-kernel is enveloped by cache-aware and memory-aware blocking layers written in a high-level language.
  - **Hardware/Software Co-design**: The optimizations are a direct response to modern CPU architecture characteristics: **deep cache hierarchies, SIMD vector units, superscalar execution, and high memory latency**. The tutorial makes these abstract concepts concrete.
- **Bridge to Production Libraries**: By demystifying the techniques behind **GotoBLAS, OpenBLAS, and BLIS**, it provides invaluable insight for developers who need to understand, tune, or extend these critical libraries. It moves optimization from "black magic" to a teachable engi-

neering discipline.

- **Educational Framework for Auto-Tuning**: The principles learned here are directly applicable to creating **auto-tuners**, which search over block sizes (`MR`, `NR`, `KC`, `MC`, `NC`) and other parameters to find the optimal configuration for a specific hardware platform, a common practice in modern linear algebra libraries.

Source Code Implementation

```
# How To Optimize Gemm wiki pages
https://github.com/flame/how-to-optimize-gemm/wiki

Copyright by Prof. Robert van de Geijn (rvdg@cs.utexas.edu).

Adapted to Github Markdown Wiki by Jianyu Huang (jianyu@cs.utexas.edu).

# Table of contents

  * [The GotoBLAS/BLIS Approach to Optimizing Matrix-Matrix Multiplication - Step-by-Step](
  * [NOTICE ON ACADEMIC HONESTY](../../wiki#notice-on-academic-honesty)
  * [References](../../wiki#references)
  * [Set Up](../../wiki#set-up)
  * [Step-by-step optimizations](../../wiki#step-by-step-optimizations)
  * [Computing four elements of C at a time](../../wiki#computing-four-elements-of-c-at-a-ti
    * [Hiding computation in a subroutine](../../wiki#hiding-computation-in-a-subroutine)
    * [Computing four elements at a time](../../wiki#computing-four-elements-at-a-time)
    * [Further optimizing](../../wiki#further-optimizing)
  * [Computing a 4 x 4 block of C at a time](../../wiki#computing-a-4-x-4-block-of-c-at-a-ti
    * [Repeating the same optimizations](../../wiki#repeating-the-same-optimizations)
    * [Further optimizing](../../wiki#further-optimizing-1)
    * [Blocking to maintain performance](../../wiki#blocking-to-maintain-performance)
    * [Packing into contiguous memory](../../wiki#packing-into-contiguous-memory)
  * [Acknowledgement](../../wiki#acknowledgement)

# Related Links
* [BLISlab: A Sandbox for Optimizing GEMM](https://github.com/flame/blislab)
* [GEMM: From Pure C to SSE Optimized Micro Kernels](http://apfel.mathematik.uni-ulm.de/~lel

# Acknowledgement
This material was partially sponsored by grants from the National Science Foundation (Awards

_Any opinions, findings and conclusions or recommendations expressed in this material are th
```

Chapter 2: Infrastructure

File: PlotAll.py

1. Role & Purpose

This file serves as a **performance visualization and comparison tool** within an HPC matrix multiplication optimization project. Its primary functions are:

- **Performance Data Extraction**: Parses custom-formatted output files from two different versions of matrix multiplication kernels
- **Visual Benchmarking**: Generates comparative plots showing GFLOPS/sec versus matrix size for both implementations
- **Peak Performance Contextualization**: Overlays theoretical hardware performance limits to evaluate optimization effectiveness
- **Regression Testing**: Facilitates visual regression analysis by comparing "old" versus "new" implementations

The script is a **post-processing analysis tool** that transforms raw timing data into actionable performance insights, enabling developers to quickly assess optimization improvements across varying problem sizes.

2. Technical Details

Parser Class Architecture

The custom `Parser` class implements a **token-based recursive descent parser** for a domain-specific language:

- **Tokenization**: Splits input files by whitespace using `file.read().split()`
- **Grammar Handling**:
  - Variables: Identifiers followed by `=` assignment
  - Values: Two supported types:
    - Lists: Square-bracket enclosed float arrays `[1.0 2.0 3.0]`
    - Strings: Single-quoted text `'version_string'`
- **Attribute Access**: Implements `__getattr__` to enable dot-notation access to parsed variables (e.g., `old.version`)

Performance Data Structure

The parsed `MY_MMult` data follows a specific **three-column format**: - **Column 0**: Matrix dimension `m = n = k` (assuming square matrices) - **Column 1**: Measured performance in GFLOPS/sec - **Column 2**: Typically contains timing or auxiliary data (not plotted)

The reshaping `reshape(-1, 3)` ensures robust handling of variable-length data by grouping elements into triplets.

Plotting Logic

The visualization employs **matplotlib** with specific design choices: - **Line Styles**: `'bo-.'` (blue circles with dash-dot) for old version, `'r-*'` (red stars with solid line) for new version - **Axes Configuration**: - X-axis: Matrix size (logarithmic scaling implied by uniform spacing) - Y-axis: GFLOPS/sec with explicit range [0, theoretical_peak] - **Dynamic Labeling**: Automatically incorporates version strings from parsed data - **Output**: Saves to `"test.png"` while also displaying interactively via `plt.show()`

Peak Performance Calculation

The theoretical peak GFLOPS is computed as:

```
max_gflops = nflops_per_cycle * nprocessors * GHz_of_processor
```

Where: - **nflops_per_cycle**: Hardware-specific FLOPs/cycle capability (4 here, typical for 2× FMA units × double-precision) - **nprocessors**: Core count for parallel execution (1 for single-core analysis) - **GHz_of_processor**: Base clock frequency (2.0 GHz)

3. HPC Context

Performance Portability Analysis

This visualization is crucial for **cross-architecture performance evaluation**: - **Architectural Bounds**: The peak GFLOPS line represents the **Roofline Model's** ridge point, helping identify if implementations are compute-bound or memory-bound - **Strong Scaling Assessment**: With `nprocessors=1`, it evaluates single-core efficiency before multi-core parallelization - **Vectorization Effectiveness**: The `nflops_per_cycle=4` suggests expectation of **AVX2** or similar SIMD utilization (2×256-bit FMA units × 2 FLOPs/cycle)

Optimization Workflow Integration

- **Kernel Tuning Validation**: Enables rapid A/B testing of optimization techniques (blocking, prefetching, register tiling)
- **Problem Size Characterization**: Reveals performance cliffs at cache boundaries (L1, L2, L3 transitions)
- **Regression Prevention**: Visual comparison ensures new optimizations don't degrade performance at specific matrix sizes

Memory Hierarchy Insights

The plot's shape provides implicit information about: - **Cache-aware algorithms**: Performance plateaus indicate effective cache utilization - **TLB effects**: Sudden drops may reveal translation lookaside buffer thrashing - **Memory bandwidth saturation**: As matrix sizes increase, performance may approach memory bandwidth limits rather than compute limits

Educational Value in HPC Pedagogy

This script demonstrates **empirical performance analysis** best practices: - **Normalized Metrics**: GFLOPS/sec enables cross-platform comparison - **Theoretical Bounding**: Contextualizes achieved performance against hardware limits - **Version Control**: Explicit labeling facilitates historical performance tracking - **Automation**: Script-based analysis ensures reproducibility in performance tuning

The visualization's clear distinction between old/new implementations with theoretical maxima provides immediate feedback on optimization effectiveness—a critical capability in the iterative process of HPC kernel development where **performance regression detection** is as important as peak performance achievement.

Source Code Implementation

```
import matplotlib.pyplot as plt
import numpy as np

# Indicate the number of floating point operations that can be executed
# per clock cycle
nflops_per_cycle = 4

# Indicate the number of processors being used (in case you are using a
# multicore or SMP)
nprocessors = 1

# Indicate the clock speed of the processor.  On a Linux machine this info
# can be found in the file /proc/cpuinfo
#
# Note: some processors have a "turbo boost" mode, which increases
# the peak clock rate...
#
GHz_of_processor = 2.0


class Parser:
    def __init__(self, file_name) -> None:
        self.attrs = {}
        with open(file_name) as file:
            self.toks = file.read().split()
            self.toksi = 0
            file.close()
            self.attrs = self.parse()

    def next(self):
```

```
        tok = self.toks[self.toksi]
        self.toksi += 1
        return tok

    def get_var_name(self):
        return self.next()

    def get_symbol(self, sym):
        tok = self.next()
        assert(tok == sym)
        return tok

    def get_value(self):
        value = None
        tok = self.next()
        if tok == '[':
            # list
            value = []
            tok = self.next()
            while not tok.startswith(']'):
                value.append(float(tok))
                tok = self.next()
        elif tok.startswith("'"):
            value = tok[1:-2]

        assert value != None
        return value

    def parse(self):
        res = {}
        while self.toksi < len(self.toks):
            var = self.get_var_name()
            self.get_symbol('=')
            val = self.get_value()
            res[var] = val
        return res

    def __getattr__(self, name):
        return self.attrs[name]

old = Parser("output_old.m")
new = Parser("output_new.m")

#print(old)
#print(new)
```

```
old_data = np.array(old.MY_MMult).reshape(-1, 3)
new_data = np.array(new.MY_MMult).reshape(-1, 3)

max_gflops = nflops_per_cycle * nprocessors * GHz_of_processor;

fig, ax = plt.subplots()
ax.plot(old_data[:,0], old_data[:,1], 'bo-.', label='old:' + old.version)
ax.plot(new_data[:,0], new_data[:,1], 'r-*', label='new:' + new.version)

ax.set(xlabel='m = n = k', ylabel='GFLOPS/sec.',
       title="OLD = {}, NEW = {}".format(old.version, new.version))
ax.grid()
ax.legend()

ax.set_xlim([old_data[0,0], old_data[-1,0]])
ax.set_ylim([0, max_gflops])

# fig.savefig("test.png")
plt.show()
```

File: REF_MMult.c

Analysis of REF_MMult.c - Reference Matrix Multiplication Implementation

1. Role & Purpose

This file serves as the **reference implementation** of matrix multiplication in the optimization study. Its primary roles are:

- **Baseline Comparison**: Provides a straightforward, unoptimized implementation against which all optimized versions can be compared for both correctness and performance.
- **Correctness Verification**: Acts as the "ground truth" implementation that generates expected results when testing optimized versions.
- **Conceptual Foundation**: Demonstrates the fundamental triple-nested loop structure of matrix multiplication ($C = A \times B + C$) that all optimizations build upon.
- **Performance Benchmark**: Establishes a baseline performance metric (typically poor due to cache inefficiency) that optimization techniques aim to improve.

2. Technical Details

Matrix Storage and Access

The implementation uses **column-major ordering**, consistent with mathematical libraries like BLAS and LAPACK:

```
#define A(i,j) a[ (j)*lda + (i) ]  // Column j, row i
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]
```

Key concepts: - `lda`, `ldb`, `ldc` are **leading dimensions** - the number of memory locations between consecutive elements in the same column - For column-major: Element at row `i`, column `j` = base pointer + `j`*ld* + *`i`* - *This differs from row-major (C/C++ native arrays) where indexing would be* *`i`*n + `j`

Algorithm Structure

The core computation follows the **ijk loop ordering**:

```
for (i=0; i<m; i++)        // Loop over rows of C and A
  for (j=0; j<n; j++)      // Loop over columns of C and B
    for (p=0; p<k; p++)    // Inner product accumulation
      C(i,j) += A(i,p) * B(p,j);
```

Mathematical interpretation: Each element C(i,j) computes the dot product of row i of A and column j of B:

```
C(i,j) = Σ A(i,p) × B(p,j) for p = 0 to k-1
```

Memory Access Pattern Analysis

- **A accesses**: `A(i,p)` - Fixed row i, varying p → **stride-1 access** in column-major (good for cache)
- **B accesses**: `B(p,j)` - Fixed column j, varying p → **stride-ldb access** (poor for cache)
- **C accesses**: `C(i,j)` - Single element per inner loop (reused across p iterations)

Critical insight: The innermost loop accesses B with large stride (`ldb`), causing **cache thrashing** as different cache lines are loaded for each p iteration.

3. HPC Context

Performance Characteristics

This implementation exhibits **suboptimal cache utilization** due to:

1. **Poor Spatial Locality**: The access pattern for matrix B (`B(p,j)`) jumps by `ldb` elements between consecutive inner loop iterations, preventing effective cache line utilization.

2. **No Data Reuse**: Each element of A is used only once per outer j-loop iteration, and each element of B is used only once per outer i-loop iteration, resulting in **O(m×n×k) memory accesses**.

3. **Memory Bandwidth Bound**: The arithmetic intensity (flops/byte) is extremely low:

   - Operations: 2 flops per multiply-add
   - Memory accesses: 3 loads/stores (A, B, C)
   - Arithmetic intensity 0.67 flops/byte (assuming 8-byte doubles)

Why This Implementation Matters for Optimization Studies

1. **Demonstrates the Memory Wall Problem**: Shows how naive implementations are limited by memory bandwidth rather than CPU compute capability.

2. **Illustrates Optimization Opportunities**:

   - **Loop reordering** (ikj, jik, etc.) to improve cache locality
   - **Blocking/tiling** to exploit temporal locality
   - **Vectorization** opportunities in the inner loop
   - **Parallelization** potential across i or j loops

3. **Serves as Performance Baseline**: Typical performance characteristics:

   - Achieves only 5-15% of peak CPU FLOPS
   - Limited by main memory bandwidth
   - Performance degrades with larger matrices due to cache misses

4. **Educational Value**: Clearly shows the **computational complexity** (O(m×n×k)) and **memory access pattern** that all optimizations must preserve while improving efficiency.

Relationship to BLAS

This implements the **GEMM** (General Matrix Multiply) operation from the BLAS Level 3 specification:

`C ← AB + C`

(Here with =1, =1, but easily extended)

The reference implementation helps understand why optimized BLAS implementations (OpenBLAS, Intel MKL, etc.) can achieve 10-50× speedups through: - Architecture-specific tuning - Cache-aware blocking algorithms - SIMD vectorization - Multi-threaded parallelization

In summary, this file represents the starting point for understanding matrix multiplication optimization—a simple, correct implementation whose poor performance motivates the need for sophisticated HPC techniques.

Source Code Implementation

```
/* Create macros so that the matrices are stored in column-major order */

#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Routine for computing C = A * B + C */

void REF_MMult( int m, int n, int k, double *a, int lda,
                                     double *b, int ldb,
                                     double *c, int ldc )
{
  int i, j, p;

  for ( i=0; i<m; i++ ){
    for ( j=0; j<n; j++ ){
      for ( p=0; p<k; p++ ){
    C( i,j ) = C( i,j ) +  A( i,p ) * B( p,j );
      }
    }
  }
}
```

File: compare_matrices.c

Analysis of `compare_matrices.c`

1. Role & Purpose This file provides a **verification utility** for high-performance matrix multiplication kernels. Its primary role is to **quantify the numerical accuracy** of optimized implementations by computing the **maximum absolute element-wise difference** between two matrices. In HPC projects, such a function is essential for:

- **Correctness validation**: Comparing output from optimized kernels (e.g., using AVX, OpenMP) against a trusted reference implementation.
- **Regression testing**: Ensuring code modifications do not introduce numerical errors.
- **Tolerance checking**: Determining if floating-point discrepancies are within acceptable limits (e.g., due to reordering of operations).

The function operates on matrices stored in **column-major order**, aligning

with the BLAS (Basic Linear Algebra Subprograms) standard used in scientific computing.

2. Technical Details Macro Definitions: - `#define A( i, j ) a[ (j)*lda + (i) ]`: Maps 2D matrix indices to a 1D array in **column-major layout**. `lda` (leading dimension) represents the number of rows in the allocated matrix, allowing access to submatrices or padded memory. - `#define B( i, j ) b[ (j)*ldb + (i) ]`: Same mapping for the second matrix. - `#define abs( x ) ( (x) < 0.0 ? -(x) : (x) )`: Inline absolute value for `double`. **Note**: This macro evaluates its argument twice, which is safe here but can cause side-effect bugs in general (e.g., `abs(x++)`).

Function Logic:

```
double compare_matrices( int m, int n, double *a, int lda, double *b, int ldb )
```

- **Parameters**:
  - `m`, `n`: Logical rows and columns of the matrices.
  - `a`, `b`: Pointers to the first elements of the matrices.
  - `lda`, `ldb`: Leading dimensions (typically `m` for memory alignment/padding).
- **Loop Structure**:
  - Outer loop over columns (`j`), inner loop over rows (`i`). This order matches **column-major storage**, ensuring **sequential memory access** within the inner loop for better cache utilization.
- **Difference Calculation**:
  - `diff = abs( A( i,j ) - B( i,j ) )`: Computes absolute difference per element.
  - `max_diff = ( diff > max_diff ? diff : max_diff )`: Tracks the global maximum difference via a conditional update.
- **Return Value**: A single `double` representing the **infinity-norm** of the matrix difference (`||A - B||`).

Key Considerations: - **No early termination**: The function always scans entire matrices, which is necessary for exact error norm computation. - **No relative error**: Returns absolute error; users may normalize this by matrix norms if needed. - **Memory layout awareness**: Explicit leading dimensions allow comparing submatrices of larger allocated blocks.

3. HPC Context Performance Relevance: 1. **Verification Overhead**: - This function is **not performance-critical**; it runs on the order of $O(mn)$ with minimal operations per element. It is typically used offline for validation. - However, its **memory access pattern** is optimized for column-major data, avoiding unnecessary cache misses that could slow down verification after fast kernel runs.

2. **Floating-Point Considerations**:

- In HPC, optimized matrix multiplication (e.g., via **SIMD vectorization**, **loop tiling**, **parallelization**) can alter floating-point rounding due to operation reordering. This function helps **quantify such numerical deviations**.
- The **maximum absolute difference** is a strict metric; small values (e.g., < 1e-10) generally indicate correctness for double-precision computations.

3. **Integration in Performance Pipelines**:
   - Often used in conjunction with **unit tests** that compare blocked, vectorized, or parallelized implementations against a naive triple-loop reference.
   - Can be extended to log errors per iteration in iterative solvers or to validate GPU results after host-device transfers.

Educational Value: - Demonstrates **column-major indexing**, essential for interfacing with Fortran, BLAS, and LAPACK. - Illustrates a **simple reduction pattern** (max-finding) that parallels reductions used in performance-critical code (e.g., norm computations). - Highlights the **trade-off between accuracy and speed**: Optimized kernels may sacrifice bit-wise reproducibility for performance, making such verification tools indispensable.

Typical Usage Example:

```
// After calling optimized matrix multiplication C_opt = A * B
double error = compare_matrices(M, N, C_ref, M, C_opt, M);
if (error < 1e-12) {
    printf("Kernel passed validation.\n");
}
```

This function embodies the **principle of verification before acceleration**—a cornerstone of reliable HPC software development.

Source Code Implementation

```
#define A( i, j ) a[ (j)*lda + (i) ]
#define B( i, j ) b[ (j)*ldb + (i) ]
#define abs( x ) ( (x) < 0.0 ? -(x) : (x) )

double compare_matrices( int m, int n, double *a, int lda, double *b, int ldb )
{
  int i, j;
  double max_diff = 0.0, diff;

  for ( j=0; j<n; j++ )
    for ( i=0; i<m; i++ ){
      diff = abs( A( i,j ) - B( i,j ) );
      max_diff = ( diff > max_diff ? diff : max_diff );
    }
```

```
  return max_diff;
}
```

File: copy_matrix.c

Analysis of `copy_matrix.c`

1. Role & Purpose

This file provides a foundational utility function for copying a two-dimensional matrix between memory buffers in **column-major order**. In the context of an HPC project focusing on matrix multiplication optimization, this routine serves multiple infrastructure purposes:

- **Data Reorganization**: Copies matrix blocks during **cache-aware blocking** strategies (e.g., packing submatrices into contiguous buffers to optimize cache reuse).
- **Abstraction Layer**: Abstracts indexing calculations via macros, simplifying code maintenance and reducing errors in more complex kernels.
- **Baseline Implementation**: Acts as a reference for validating optimized versions (e.g., vectorized or threaded copies) and for performance comparison.

---

2. Technical Details

Macro-Based Indexing

```
#define A( i, j ) a[ (j)*lda + (i) ]
#define B( i, j ) b[ (j)*ldb + (i) ]
```

- These macros compute the linear memory index for element `(i, j)` in a **column-major** layout.
- `lda` and `ldb` are the **leading dimensions**: the number of rows allocated for each matrix (actual rows `m`). This allows copying submatrices from larger allocated arrays.
- The indexing formula `(j)*ld + (i)` reflects column-major order: elements in the same column are contiguous in memory.

Function Logic

```
void copy_matrix( int m, int n, double *a, int lda, double *b, int ldb )
{
  int i, j;
  for ( j=0; j<n; j++ )
```

```
    for ( i=0; i<m; i++ )
      B( i,j ) = A( i,j );
}
```

- **Parameters**:
  - `m`, `n`: Number of rows and columns to copy.
  - `a`, `b`: Source and destination pointers.
  - `lda`, `ldb`: Leading dimensions of source and destination.
- **Loop Structure**:
  - **Outer loop** over columns (`j`), **inner loop** over rows (`i`).
  - This order maximizes **spatial locality** for column-major storage: inner-loop iterations access contiguous memory addresses (`A(i,j)` → `A(i+1,j)`).
- **Memory Access Pattern**:
  - Source and destination are accessed with **unit stride** (contiguous in inner loop), which is cache-friendly.
  - Strides between columns are `lda` (source) and `ldb` (destination), which may differ if matrices have different allocations.

Performance Considerations in Implementation

- **No vectorization or parallelism**: The plain C loops rely on compiler optimizations.
- **Potential overhead**: The macros are expanded inline, but indexing calculations remain in the inner loop. An optimizing compiler may hoist invariants like `j*lda` and `j*ldb`.
- **Missing alignment checks**: No guarantees that `a` or `b` are aligned to SIMD boundaries (e.g., 32-byte for AVX), which can hinder vectorized optimizations.

---

3. HPC Context

Relevance to Matrix Multiplication Optimization

- **Blocking (Tiling)**: In optimized matrix multiplication (e.g., GotoBLAS, OpenBLAS), matrices are divided into small blocks that fit in L1/L2 caches. `copy_matrix` can pack these blocks into **contiguous, aligned buffers** to ensure unit-stride access and eliminate TLB misses during the core multiplication kernel.
- **Data Layout Transformations**: May be extended to copy with **transposition** or **padding** to meet alignment requirements for SIMD instructions.
- **Memory Bandwidth**: As a memory-bound operation, its performance scales with **effective bandwidth utilization**. Optimized variants using **AVX intrinsics** or **non-temporal stores** can reduce cache pollution and improve throughput.

Performance Implications

- **Cache Efficiency**: The current column-major inner loop exploits spatial locality, but performance can degrade if `lda` or `ldb` are large (causing cache thrashing). Blocked copying may be needed for very large matrices.
- **Vectorization Potential**:
  - The contiguous access pattern allows automatic vectorization by compilers (e.g., `-O3 -mavx2`).
  - Manual optimization using **AVX/AVX2 intrinsics** (e.g., `_mm256_loadu_pd`, `_mm256_storeu_pd`) could achieve near-peak bandwidth.
- **Parallelization**: For large matrices, this operation can be parallelized over columns (OpenMP) or tiled for multi-threaded execution.

Educational Value

- Illustrates **canonical column-major indexing**, a standard in BLAS and LAPACK.
- Demonstrates a **memory-bound kernel** where optimization focuses on access patterns and vectorization.
- Serves as a baseline for exploring **data layout transformations** critical in HPC (e.g., copy with stride adjustments for GPU offloading).

Key Takeaways

- The function is a **basic building block** for matrix operations, emphasizing correct indexing and access patterns.
- In production HPC libraries, similar routines are **highly optimized** using SIMD, alignment, and parallelism to minimize memory bottlenecks.
- Understanding this simple implementation is essential for grasping more advanced optimization techniques in dense linear algebra.

Source Code Implementation

```
#define A( i, j ) a[ (j)*lda + (i) ]
#define B( i, j ) b[ (j)*ldb + (i) ]

void copy_matrix( int m, int n, double *a, int lda, double *b, int ldb )
{
  int i, j;

  for ( j=0; j<n; j++ )
    for ( i=0; i<m; i++ )
      B( i,j ) = A( i,j );
}
```

File: dclock.c

Technical Analysis of `dclock.c`

1. Role & Purpose

The `dclock.c` file provides a **high-resolution timing function** designed specifically for performance benchmarking in HPC applications. It serves as a **precise wall-clock timer** that measures elapsed time with **microsecond precision**, making it essential for evaluating computational kernels like matrix multiplication. The code is **adapted from the BLIS (BLAS-like Library Instantiation Software) library**, indicating its heritage in high-performance linear algebra systems.

Key Functions in Project Infrastructure:

- **Benchmarking Core**: Forms the timing foundation for performance measurements
- **Cross-Platform Compatibility**: Uses standard POSIX timing functions
- **Reference Time Management**: Avoids issues with system clock wraparound
- **Zero-overhead Timing**: Minimal function call overhead for accurate measurements

2. Technical Details

Function Implementation Analysis:

```
double dclock()
{
    double         the_time, norm_sec;
    struct timeval tv;
```

Data Types and Structures: - `struct timeval`: POSIX structure containing `tv_sec` (seconds) and `tv_usec` (microseconds) - `double`: Floating-point type for high-precision time representation - `static double gtod_ref_time_sec`: **Static variable** maintaining state between calls

Algorithm Flow:

1. **Time Acquisition**:

   ```
   gettimeofday(&tv, NULL);
   ```

   - Calls the **POSIX `gettimeofday()`** system call

- Returns seconds and microseconds since the Unix epoch (1970-01-01)
- `NULL` parameter indicates timezone not needed

2. **Reference Time Initialization**:

```
if (gtod_ref_time_sec == 0.0)
    gtod_ref_time_sec = (double)tv.tv_sec;
```

   - **Lazy initialization**: Sets reference on first call only
   - Captures the **application start time** as baseline
   - `static` storage ensures persistence across calls

3. **Normalized Time Calculation**:

```
norm_sec = (double)tv.tv_sec - gtod_ref_time_sec;
the_time = norm_sec + tv.tv_usec * 1.0e-6;
```

   - **Subtracts reference time** to avoid large floating-point values
   - Combines seconds and microseconds into a single `double`
   - Microseconds converted using **literal floating-point multiplication**

Precision Considerations:

- **Microsecond resolution**: `tv.tv_usec` provides 10 second precision
- **Double-precision arithmetic**: Maintains accuracy over long runs
- **No system call overhead minimization**: `gettimeofday()` is relatively fast (~25-100 nanoseconds)

3. HPC Context

Performance Measurement Relevance:

1. **Kernel Timing Precision**:

   - Matrix multiplication optimizations require **sub-millisecond timing**
   - **Microsecond resolution** captures performance differences between optimization techniques
   - Essential for measuring **loop tiling, vectorization, and cache blocking** effects

2. **Benchmarking Methodology**:

   - **Minimal overhead**: Function designed to not interfere with measured code
   - **Consistent reference**: Static reference time enables **relative timing** across multiple runs
   - **Portable abstraction**: Hides platform-specific timing details from optimization code

3. **Statistical Measurement**:
    - **High resolution** enables collection of **multiple samples** for statistical analysis
    - Critical for **variance measurement** in modern CPUs with dynamic frequency scaling
    - Supports **warm-up iterations** and steady-state measurement patterns
4. **Comparison with Alternatives**:

```
// Common HPC timing alternatives:
clock_gettime(CLOCK_MONOTONIC, ...)  // ~10x more overhead
rdtsc()                              // Cycle counter, CPU-specific
omp_get_wtime()                      // OpenMP wrapper
```

    - `gettimeofday()` offers **good balance** of precision and portability
    - Less overhead than `clock_gettime()` but potentially affected by **NTP adjustments**
    - **Not monotonic** (can jump backward with time adjustments)
5. **Optimization Impact**:
    - Timing accuracy directly affects **performance model validation**
    - Enables **roofline model** construction by separating compute and memory bottlenecks
    - Critical for **auto-tuning systems** that explore parameter spaces

Limitations and Considerations:

- **System call overhead**: Each call enters kernel space (~100ns)
- **Non-monotonic**: System time adjustments can cause time to jump backward
- **Deprecated status**: `gettimeofday()` marked obsolete in POSIX.1-2008
- **Better alternative**: `clock_gettime(CLOCK_MONOTONIC)` for modern systems

Educational Value:

This implementation demonstrates **fundamental HPC benchmarking principles**: - **Isolation of measurement overhead - Reference time normalization** for long-running applications - **Trade-offs between precision, accuracy, and portability - Importance of temporal resolution** relative to operation duration

Best Practice: For matrix multiplication timing, call `dclock()` immediately before and after the computational kernel, discarding initial warm-up iterations to account for CPU frequency scaling and cache warming effects.

Source Code Implementation

```c
#include <sys/time.h>
#include <time.h>

static double gtod_ref_time_sec = 0.0;

/* Adapted from the bl2_clock() routine in the BLIS library */

double dclock()
{
        double         the_time, norm_sec;
        struct timeval tv;

        gettimeofday( &tv, NULL );

        if ( gtod_ref_time_sec == 0.0 )
                gtod_ref_time_sec = ( double ) tv.tv_sec;

        norm_sec = ( double ) tv.tv_sec - gtod_ref_time_sec;

        the_time = norm_sec + tv.tv_usec * 1.0e-6;

        return the_time;
}
```

File: makefile

1. Role & Purpose

This **Makefile** serves as the **build automation and benchmarking system** for a matrix multiplication optimization study. It orchestrates:

- **Compilation** of different matrix multiplication kernel implementations (`MMult0.c`, etc.) with performance-oriented compiler flags
- **Linking** with supporting utility functions for matrix operations and timing
- **Execution** of performance tests while controlling runtime environment variables
- **Data collection** that outputs results in MATLAB/Octave format for subsequent analysis and plotting
- **Clean management** of build artifacts and generated data files

The file enables **A/B performance comparison** between an old implementation (`OLD`) and a new implementation (`NEW`), which is crucial for iterative

optimization workflows in HPC kernel development.

2. Technical Details

Make Variables and Configuration

- `OLD/NEW`: Control which implementations are compared (initially both set to `MMult0`)
- `CC/LINKER`: Specify the GNU Compiler Collection (GCC) toolchain
- `CFLAGS`: Critical optimization flags:
  - `-O2`: Enables standard optimizations (loop unrolling, inlining)
  - `-Wall`: Shows all warnings for code quality
  - `-msse3`: Enables Streaming SIMD Extensions 3 instructions for vectorization
- `UTIL`: Object files providing essential matrix operations:
  - `copy_matrix.o`, `compare_matrices.o`: For data management and verification
  - `random_matrix.o`: Generates test data with controlled properties
  - `dclock.o`: High-resolution timing (likely using `gettimeofday()` or `clock_gettime()`)
  - `REF_MMult.o`: Reference implementation (likely naive triple-loop)
  - `print_matrix.o`: Debugging output

Build Rules and Targets

- **Pattern rule `%.o: %.c`**: Compiles C sources to objects with the specified `CFLAGS`
  - *Note: Duplicate rule is redundant but harmless*
- `test_MMult.x`: Main executable target
  - **Dependencies**: Test driver (`test_MMult.o`), new kernel (`$(NEW).o`), utilities, and `parameters.h` (likely defines matrix sizes, blocking factors)
  - **Linking**: Combines all objects with math library (`-lm`) and optional BLAS library
  - *Issue: `$(TEST_BIN)` variable undefined; should likely be `-o $@`*

Benchmark Execution Logic (`run` target)

- **Thread control**: Sets `OMP_NUM_THREADS=1` and `GOTO_NUM_THREADS=1` to ensure single-threaded execution
- **Output pipeline**:
  1. Creates MATLAB/Octave script with version metadata: `echo "version = '$(NEW)';" > output_$(NEW).m`
  2. Appends benchmark results: `./test_MMult.x >> output_$(NEW).m`
  3. Copies outputs to standardized names for comparison scripts
- **Data format**: Results in `.m` files allow direct plotting with MATLAB/Octave visualization tools

Memory Management

- **Clean targets**: `clean` removes build artifacts; `cleanall` also deletes output data and plots
- **Object files**: Separately compiled for modularity and incremental builds

3. HPC Context

Performance Isolation and Measurement

- **Single-threading enforcement**: By setting thread environment variables, the Makefile ensures performance measurements isolate **single-core algorithmic efficiency** from parallel scaling effects
- **Controlled compilation**: The `-O2 -msse3` flags represent a **balanced optimization level** that enables auto-vectorization while avoiding overly aggressive transformations that might obscure manual optimization efforts
- **Reference implementation**: Including `REF_MMult.o` provides a **baseline performance** comparison essential for calculating speedup ratios

Optimization Workflow Support

- **A/B testing infrastructure**: The `OLD/NEW` comparison mechanism enables systematic optimization progression
- **Reproducible builds**: Clean rebuilding (`make clean; make test_MMult.x`) ensures no stale objects affect performance measurements
- **Automated data collection**: The pipeline from compilation to data generation supports **rapid iteration** in the optimize-measure-analyze cycle

Microarchitecture Considerations

- **SSE3 vectorization**: The `-msse3` flag targets a specific **SIMD instruction set** prevalent in x86_64 processors, enabling:
  - **Packaged floating-point operations**: 4 single-precision or 2 double-precision operations per instruction
  - **Horizontal operations**: Useful for reduction patterns in matrix multiplication
  - **Improved data movement**: Specialized load/store instructions
- **Cache-aware benchmarking**: The test driver (not shown) likely sweeps through matrix sizes to measure performance across **cache hierarchy regimes** (L1, L2, L3, TLB)

Performance Engineering Practices

- **Timing discipline**: `dclock.o` suggests high-resolution timing, crucial for measuring short-duration kernel performance

- **Correctness verification**: `compare_matrices.o` ensures algorithmic changes maintain numerical correctness
- **Data-oriented output**: MATLAB/Octave formatted results facilitate **performance visualization** (roofline plots, cache effect graphs)

This Makefile embodies **best practices in computational kernel development**: automated builds, controlled experimentation environments, and integrated performance data collection—all essential for systematic HPC optimization.

Source Code Implementation

```
OLD  := MMult0
NEW  := MMult0
#
# sample makefile
#

CC         := gcc
LINKER     := $(CC)
CFLAGS     := -O2 -Wall -msse3
LDFLAGS    := -lm

UTIL       := copy_matrix.o \
              compare_matrices.o \
              random_matrix.o \
              dclock.o \
              REF_MMult.o \
              print_matrix.o

TEST_OBJS  := test_MMult.o $(NEW).o

%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

all:
    make clean;
    make test_MMult.x

test_MMult.x: $(TEST_OBJS) $(UTIL) parameters.h
    $(LINKER) $(TEST_OBJS) $(UTIL) $(LDFLAGS) \
        $(BLAS_LIB) -o $(TEST_BIN) $@

run:
```

```
    make all
    export OMP_NUM_THREADS=1
    export GOTO_NUM_THREADS=1
    echo "version = '$(NEW)';" > output_$(NEW).m
    ./test_MMult.x >> output_$(NEW).m
    cp output_$(OLD).m output_old.m
    cp output_$(NEW).m output_new.m

clean:
    rm -f *.o *~ core *.x

cleanall:
    rm -f *.o *~ core *.x output*.m *.eps *.png
```

File: parameters.h

Analysis of `parameters.h`

1. Role & Purpose

This header file serves as a **configuration interface** for a matrix multiplication benchmarking suite. It parameterizes experimental sweeps across problem dimensions and controls memory layout, enabling systematic performance studies. In the project infrastructure, it decouples test configuration from implementation, allowing users to modify experimental setups without altering core computational kernels. The parameters govern: - **Problem size ranges** through a looping variable `p` - **Matrix dimensions** (M, N, K) that may be statically fixed or dynamically scaled - **Statistical reliability** via repeated trials - **Memory layout** through leading dimension specifications

2. Technical Details

The file defines C preprocessor macros that control a test driver's execution:

- **Loop Control Macros** (`PFIRST`, `PLAST`, `PINC`):
  - Create a parameter sweep: `for(p = PFIRST; p <= PLAST; p += PINC)`
  - Example: With given values, `p` takes values 40, 80, ..., 800
  - This sweep enables scaling studies to observe performance transitions
- **Dimension Binding Macros** (`M`, `N`, `K`):
  - When set to `-1`, dimensions bind to the loop index `p`
  - With `M = N = K = -1`, all dimensions equal `p` → **square matrix multiplication**
  - Alternative configurations:
    - Fixed dimensions: `M = 100, N = 200, K = 300`

    - Mixed: `M = -1, N = 100, K = -1` → variable M and K, fixed N
- **Experimental Repetition** (`NREPEATS`):
  - Each `(M,N,K)` configuration runs `NREPEATS` times
  - **Best-of-N timing** minimizes measurement noise from system variability
  - Critical for reliable performance measurements in shared environments
- **Leading Dimension Macros** (`LDA`, `LDB`, `LDC`):
  - Control **memory stride** in row-major storage: `A[i][j]` → `A[i*LDA + j]`
  - When `LDX = -1`: Leading dimension = row dimension (contiguous rows)
  - With `LDX = 1000`: Rows are **padded**, creating **stride access patterns**
  - Memory layout example for `M = 800`:
    - Row elements: `A[i][0]` to `A[i][799]`
    - Memory gap: Elements `A[i][800]` to `A[i][999]` are allocated but unused
    - Next row starts at `A[i+1][0]` = `A[i][0] + 1000*sizeof(double)`

3. HPC Context

These parameters directly impact performance analysis in three critical ways:

- **Performance Scaling Studies**: The `p`-loop enables observation of **cache hierarchy effects** and **regime transitions**. Small matrices fit in cache, while larger ones exercise memory bandwidth. The 40–800 range captures L1/L2/L3 cache boundaries on typical CPUs.
- **Memory Hierarchy Optimization**: **Leading dimensions** control **spatial locality** and **cache line utilization**:
  - When `LDX > M`: **Strided access** causes **cache thrashing** and **TLB pressure** due to unused data fetched into cache lines
  - Optimal performance typically requires `LDX = M` (no padding) for contiguous access
  - The fixed `LDX=1000` creates artificial **cache contention** scenarios for performance analysis
- **Statistical Measurement Integrity**:
  - `NREPEATS=2` provides minimal averaging while mitigating **system noise** (OS interrupts, power management, etc.)
  - **Best-of-N timing** avoids outliers from cold starts or context switches
  - In production benchmarks, `NREPEATS` is often increased (5-10) for statistical significance

- **Architectural Implications**: The configuration tests **SIMD vectorization efficiency** across different strides. With `LDX=1000`, vector loads may span multiple cache lines (**cache line splits**), degrading AVX/AVX2/AVX-512 performance. This models real-world scenarios where matrices embed within larger data structures.

Educational Insight: This configuration file embodies key HPC principles: **parameterized experimentation**, **memory hierarchy awareness**, and **statistical rigor**. By varying these parameters, students can empirically observe how algorithmic complexity interacts with hardware constraints—a fundamental skill in performance engineering.

Source Code Implementation

```
/*
In the test driver, there is a loop "for ( p=PFIRST; p<= PLAST; p+= PINC )"
The below parameters set this range of values that p takes on
*/
#define PFIRST 40
#define PLAST  800
#define PINC   40

/*
In the test driver, the m, n, and k dimensions are set to the below
values.  If the value equals "-1" then that dimension is bound to the
index p, given above.
*/

#define M -1
#define N -1
#define K -1

/*
In the test driver, each experiment is repeated NREPEATS times and
the best time from these repeats is used to compute the performance
*/

#define NREPEATS 2

/*
Matrices A, B, and C are stored in two dimensional arrays with
row dimensions that are greater than or equal to the row dimension
of the matrix.  This row dimension of the array is known as the
"leading dimension" and determines the stride (the number of
double precision numbers) when one goes from one element in a row
to the next.  Having this number larger than the row dimension of
```

```
the matrix tends to adversely affect performance.  LDX equals the
leading dimension of the array that stores matrix X.  If LDX=-1
then the leading dimension is set to the row dimension of matrix X.
*/

#define LDA 1000
#define LDB 1000
#define LDC 1000
```

File: print_matrix.c

Analysis of `print_matrix.c`

1. Role & Purpose This file provides a **debugging utility** for inspecting matrix data stored in **column-major order**, which is the standard memory layout in linear algebra libraries like **BLAS** and **LAPACK**. The function `print_matrix` outputs the matrix contents in a human-readable format, typically used for **verifying correctness** during development, testing, or benchmarking of high-performance linear algebra kernels. It allows developers to confirm that matrix data—whether representing inputs, intermediates, or results—is correctly populated, especially after complex transformations or operations.

---

2. Technical Details

- **Function Signature**:
  `void print_matrix(int m, int n, double *a, int lda)`
  - `m`: Number of rows in the matrix to print.
  - `n`: Number of columns.
  - `a`: Pointer to the matrix data in **column-major order**.
  - `lda`: **Leading dimension** of the matrix (`m`). This is the stride between consecutive rows in memory for a given column, enabling the function to handle both **full matrices** and **submatrices** (e.g., tiles or blocks within a larger array).
- **Indexing Macro**:
  `#define A(i, j) a[(j)*lda + (i)]`
  This macro computes the memory offset for element `(i, j)` using column-major addressing:
  - Each column `j` is stored contiguously, with `lda` elements separating the start of column `j` and column `j+1`.

- The row index `i` selects the element within column `j`.

- Parentheses around `i` and `j` prevent precedence issues if expressions are passed.

- **Printing Logic**:
  The nested loops iterate:
  1. **Outer loop over columns** (`j = 0` to `n-1`).

  2. **Inner loop over rows** (`i = 0` to `m-1`).
     Each element is printed using `%le` (scientific notation, double precision).
     Critical nuance: The output is **transposed relative to conventional row-wise display**—each line corresponds to a **full column** of the original matrix. For a 3×2 matrix, the output appears as:

```
A(0,0) A(1,0) A(2,0)
A(0,1) A(1,1) A(2,1)
```

  This reflects the **column-major storage** but may be confusing if row-wise interpretation is intended.

---

3. HPC Context

- **Memory Layout Awareness**:
  The explicit use of `lda` and column-major indexing is essential in HPC for **efficient memory access patterns**. Many optimized linear algebra kernels (e.g., matrix multiplication) rely on **column-major ordering** to leverage **spatial locality** and **vectorized loads/stores**. This function helps validate that data is correctly arranged for such kernels.

- **Submatrix Handling**:
  In **blocked algorithms** (e.g., cache-aware tiling in GEMM), matrices are often partitioned into **submatrices** with potentially larger `lda` than `m`. The function correctly accesses these submatrices by using `lda` as the stride, ensuring that debugging can be performed on **tiles without data rearrangement**.

- **Performance Implications**:
  While this function is not performance-critical, its design avoids **unnecessary transposition** during printing, which would be computationally expensive for large matrices. By directly traversing memory in **storage order** (column-by-column), it minimizes cache misses and respects the **memory hierarchy** even in a debugging context.

- **Debugging Optimized Kernels**:
  When implementing **SIMD-vectorized (e.g., AVX) or multi-threaded matrix operations**, verifying intermediate results is

challenging. This utility allows **spot-checking** of matrix blocks after key computational steps, ensuring that **vectorization masks**, **thread synchronization**, or **boundary conditions** are correctly handled.

Summary

`print_matrix.c` is a **specialized debugging tool** tailored to column-major matrix storage, enabling precise inspection of data in HPC workflows. Its design aligns with the **memory access patterns** of high-performance linear algebra, supporting both full matrices and submatrices through explicit leading dimension handling. While simple, it plays a crucial role in the **development and verification** of optimized computational kernels.

Source Code Implementation

```c
#include <stdio.h>

#define A( i, j ) a[ (j)*lda + (i) ]

void print_matrix( int m, int n, double *a, int lda )
{
  int i, j;

  for ( j=0; j<n; j++ ){
    for ( i=0; i<m; i++ )
      printf("%le ", A( i,j ) );
    printf("\n");
  }
  printf("\n");
}
```

File: random_matrix.c

Analysis of `random_matrix.c`

1. Role & Purpose

This file implements a **random matrix generator** function designed for benchmarking and testing numerical algorithms in High Performance Computing (HPC). Its primary roles are:

- **Test Data Generation**: Creates matrices populated with pseudo-random double-precision floating-point numbers uniformly distributed in

the range [-1.0, 1.0).
- **Algorithm Validation**: Provides input matrices for verifying correctness of linear algebra operations (particularly matrix multiplication).
- **Performance Benchmarking**: Supplies consistent, reproducible test cases for measuring computational performance across different optimization strategies.
- **Project Infrastructure**: Serves as a **test harness component** that isolates random number generation from computational kernels, enabling fair performance comparisons.

2. Technical Details

Function Signature and Parameters

```
void random_matrix(int m, int n, double *a, int lda)
```

- `m`: Number of matrix rows
- `n`: Number of matrix columns

- `a`: Pointer to matrix storage (1D array)
- `lda`: Leading dimension - the stride between columns in memory

Memory Access Pattern and Macro

```
#define A(i,j) a[(j)*lda + (i)]
```

- **Column-major ordering**: This is the standard storage format for BLAS (Basic Linear Algebra Subprograms) and LAPACK. Element `(i,j)` is stored at offset `j*lda + i`.
- **`lda` usage**: Allows working with submatrices or matrices with padding for alignment. Must satisfy `lda  m`.

Random Number Generation

```
double drand48();
A(i,j) = 2.0 * drand48() - 1.0;
```

- **`drand48()`**: Returns uniformly distributed pseudo-random doubles in `[0.0, 1.0)`
- **Linear transformation**: `2.0 * drand48() - 1.0` maps to `[-1.0, 1.0)`
- **Numerical properties**: Zero-centered distribution helps avoid overflow in subsequent computations.

Loop Structure and Memory Access

```
for (j = 0; j < n; j++)
    for (i = 0; i < m; i++)
        A(i,j) = ...;
```

- **Column-wise traversal**: Outer loop over columns, inner loop over rows
- **Unit stride access**: Inner loop accesses contiguous memory locations (`i` increments by 1), enabling **spatial locality** and potential **vectorization**
- **Cache-friendly**: Sequential memory access pattern minimizes cache misses

Portability Considerations

- `drand48()` is POSIX standard but not C89/C99 standard
- For strict portability, consider `rand()` with scaling or C++11 `<random>`
- Random seed management is external to this function

3. HPC Context

Performance Implications

1. **Memory Access Pattern**: The column-major, unit-stride access is **cache-optimal** for subsequent column-major operations. However, this pattern may cause **performance asymmetry** when interfacing with row-major code (common in C/C++ applications without BLAS).

2. **Random Number Generation Bottleneck**:
   - `drand48()` uses 48-bit integer arithmetic with modular multiplication
   - For very large matrices, random generation can become a **non-trivial overhead**
   - In performance-critical benchmarks, consider pre-generating random data or using faster PRNGs like **Xorshift** or **PCG**

3. **Memory Alignment Considerations**:

   ```
   // Missing alignment handling - potential optimization:
   void random_matrix_aligned(int m, int n, double *a, int lda) {
       // Handle initial misaligned elements separately
       // Use vectorized operations for aligned sections
   }
   ```

   The current implementation doesn't guarantee **SIMD-aligned accesses**, potentially limiting vectorization opportunities on architectures requiring aligned loads/stores (like some AVX instructions).

4. **Reproducibility vs. Performance**:
   - `drand48()` provides deterministic sequences given the same seed
   - For parallel implementations, thread-safe random generation would be needed
   - Consider **counter-based PRNGs** (like Philox) for parallel reproducibility

5. **Numerical Considerations for Testing**:

- Uniform distribution in `[-1, 1)` avoids pathological cases (like all zeros or huge values)
- Symmetric range helps catch sign-related bugs
- For stress testing, consider adding controlled **ill-conditioned matrices**

Optimization Opportunities

1. **Vectorization**: The inner loop is trivially vectorizable. With compiler flags (`-O3 -march=native`), modern compilers can generate **AVX/AVX2 instructions** for the assignment.
2. **Parallelization**: The outer loop is **embarrassingly parallel** with OpenMP:

   ```
   #pragma omp parallel for
   for (j = 0; j < n; j++) {
       for (i = 0; i < m; i++) {
           A(i,j) = 2.0 * drand48() - 1.0;
       }
   }
   ```

   (Note: `drand48()` is not thread-safe; need thread-local PRNG state)
3. **Streaming Stores**: For very large matrices that won't fit in cache, **non-temporal stores** could bypass cache to avoid pollution:

   ```
   #include <immintrin.h>
   _mm256_stream_pd(&A(i,j), _mm256_set1_pd(value));
   ```
4. **Blocking for Very Large Matrices**: For extreme-scale matrices, consider generating in blocks to improve **TLB (Translation Lookaside Buffer)** efficiency.

Integration in HPC Benchmarking Pipeline

This function typically feeds into a benchmarking workflow:

`random_matrix() → compute_kernel() → validate() → profile()`

The simplicity and predictability of this generator makes it ideal for **A/B testing** different optimization strategies while controlling for input data variability.

Alternative Approaches in Production HPC

- **Memory-mapped files**: For matrices larger than system memory
- **Parallel I/O patterns**: When generating distributed matrices across nodes
- **Compressed random generation**: Using random seeds and deterministic algorithms to generate matrix elements on-demand without storage

This implementation represents a **baseline approach** that balances simplicity with reasonable performance, suitable for pedagogical contexts and initial benchmarking before moving to more sophisticated generation strategies in production HPC environments.

Source Code Implementation

```c
#include <stdlib.h>

#define A( i,j ) a[ (j)*lda + (i) ]

void random_matrix( int m, int n, double *a, int lda )
{
  double drand48();
  int i,j;

  for ( j=0; j<n; j++ )
    for ( i=0; i<m; i++ )
      A( i,j ) = 2.0 * drand48( ) - 1.0;
}
```

File: test_MMult.c

1. Role & Purpose

This file serves as the **primary benchmarking driver** for evaluating matrix multiplication implementations in an HPC optimization project. Its core purposes are:

- **Performance Measurement**: Systematically tests matrix multiplication implementations across varying problem sizes while measuring execution time and calculating **GFLOPS** (Giga-FLOating-point Operations Per Second).
- **Correctness Verification**: Compares optimized implementations against a reference implementation to detect numerical errors introduced by optimization techniques.
- **Data Generation**: Produces structured output (formatted as a MATLAB/Octave matrix) suitable for performance plotting and analysis.
- **Infrastructure Integration**: Provides a standardized test harness that isolates the computational kernel from benchmarking logic, enabling fair comparison between different optimization strategies.

2. Technical Details

Key Program Flow

1. **Parameter-Driven Loop**: Iterates through problem sizes defined in `parameters.h` (from `PFIRST` to `PLAST` with increment `PINC`). These parameters allow systematic exploration of performance across different matrix dimensions.

2. **Matrix Configuration**:
   - Dimensions `m`, `n`, `k` are determined by either fixed values (`M`, `N`, `K`) or set equal to the loop variable `p`.
   - **Leading dimensions** (`lda`, `ldb`, `ldc`) control memory stride, enabling testing of both packed and padded memory layouts.
   - GFLOPS calculation: `2*m*n*k*1e-9` (since matrix multiplication requires 2 floating-point operations per multiply-add).

3. **Memory Management**:
   - Allocates five matrices using `malloc()`:
     - `a`: Input matrix A with intentional overallocation (`k+1` columns) to prevent **prefetching-related segmentation faults**
     - `b`: Input matrix B
     - `c`: Output matrix for optimized implementation
     - `cold`: Original copy of output matrix (preserved for timing)
     - `cref`: Reference output matrix
   - Uses **column-major ordering** (typical for BLAS/LAPACK), where `lda` represents the distance between elements in the same column.

4. **Benchmarking Methodology**:
   - **Reference execution**: `REF_MMult()` computes baseline result for correctness checking
   - **Timing loop**: Runs `MY_MMult()` multiple times (`NREPEATS`) and records the **minimum execution time** via `dclock()` (high-resolution timer)
   - **Statistical approach**: Taking the minimum time reduces variance from system noise (context switches, interrupts, etc.)
   - **Correctness check**: `compare_matrices()` computes the difference between optimized and reference results (typically Frobenius norm)

5. **Output Format**:

```
MY_MMult = [
p  GFLOPS  diff
p  GFLOPS  diff
...
];
```

This format enables direct plotting in MATLAB/Octave for performance visualization.

Critical Implementation Details

- **Pointer arithmetic**: Matrix elements are accessed as `a[i + j*lda]` for row `i`, column `j`
- **Memory alignment**: The overallocation in matrix A prevents false sharing and cache conflicts that could occur with certain SIMD optimizations
- **Warm-up exclusion**: The first iteration's time isn't used for `dtime_best` initialization, ensuring cache-warmed measurements

3. HPC Context

Performance Evaluation Significance

This benchmarking infrastructure is essential for HPC optimization because:

1. **Isolates Optimization Impact**: By fixing all variables except the computation kernel, it accurately measures the effect of specific optimizations (loop tiling, SIMD vectorization, register blocking, etc.)
2. **Explores Performance Regimes**: The parameter sweep reveals how optimizations perform across:
   - **L1/L2 cache-bound regimes** (small matrices)
   - **TLB and memory bandwidth-bound regimes** (large matrices)
   - **Kernel-dominated regimes** (square matrices near cache capacity)
3. **Validates Numerical Stability**: The difference check (`diff`) is crucial because:
   - Aggressive optimizations (like fused multiply-add reordering) can change floating-point rounding
   - SIMD reductions may produce different summation orders
   - Blocking strategies can alter computation sequence
4. **Standardized Measurement**: The **minimum time** approach follows best practices in HPC benchmarking by:
   - Reducing measurement noise from OS interference
   - Providing reproducible performance metrics
   - Avoiding statistical distortions from outliers

Architectural Considerations

The benchmark design accounts for modern CPU features: - **Prefetching**: The extra column in matrix A prevents speculative loads from causing segmentation faults - **Cache effects**: Multiple repetitions help account for cold vs warm

cache scenarios - **Vectorization readiness**: The structured loops and memory layouts enable compiler auto-vectorization and manual intrinsic usage

Pedagogical Value

This file demonstrates **key HPC benchmarking principles**: - Separation of computation from measurement - Statistical handling of timing variability - Correctness verification alongside performance measurement - Parameterized exploration of algorithm behavior

The infrastructure enables systematic optimization by providing consistent, reliable feedback on both performance and accuracy—allowing developers to iterate through optimization techniques while maintaining numerical correctness.

Source Code Implementation

```c
#include <stdio.h>
// #include <malloc.h>
#include <stdlib.h>

#include "parameters.h"

void REF_MMult(int, int, int, double *, int, double *, int, double *, int );
void MY_MMult(int, int, int, double *, int, double *, int, double *, int );
void copy_matrix(int, int, double *, int, double *, int );
void random_matrix(int, int, double *, int);
double compare_matrices( int, int, double *, int, double *, int );

double dclock();

int main()
{
  int
    p,
    m, n, k,
    lda, ldb, ldc,
    rep;

  double
    dtime, dtime_best,
    gflops,
    diff;

  double
    *a, *b, *c, *cref, *cold;

  printf( "MY_MMult = [\n" );
```

```
for ( p=PFIRST; p<=PLAST; p+=PINC ){
  m = ( M == -1 ? p : M );
  n = ( N == -1 ? p : N );
  k = ( K == -1 ? p : K );

  gflops = 2.0 * m * n * k * 1.0e-09;

  lda = ( LDA == -1 ? m : LDA );
  ldb = ( LDB == -1 ? k : LDB );
  ldc = ( LDC == -1 ? m : LDC );

  /* Allocate space for the matrices */
  /* Note: I create an extra column in A to make sure that
     prefetching beyond the matrix does not cause a segfault */
  a = ( double * ) malloc( lda * (k+1) * sizeof( double ) );
  b = ( double * ) malloc( ldb * n * sizeof( double ) );
  c = ( double * ) malloc( ldc * n * sizeof( double ) );
  cold = ( double * ) malloc( ldc * n * sizeof( double ) );
  cref = ( double * ) malloc( ldc * n * sizeof( double ) );

  /* Generate random matrices A, B, Cold */
  random_matrix( m, k, a, lda );
  random_matrix( k, n, b, ldb );
  random_matrix( m, n, cold, ldc );

  copy_matrix( m, n, cold, ldc, cref, ldc );

  /* Run the reference implementation so the answers can be compared */

  REF_MMult( m, n, k, a, lda, b, ldb, cref, ldc );

  /* Time the "optimized" implementation */
  for ( rep=0; rep<NREPEATS; rep++ ){
    copy_matrix( m, n, cold, ldc, c, ldc );

    /* Time your implementation */
    dtime = dclock();

    MY_MMult( m, n, k, a, lda, b, ldb, c, ldc );

    dtime = dclock() - dtime;

    if ( rep==0 )
  dtime_best = dtime;
    else
```

```
    dtime_best = ( dtime < dtime_best ? dtime : dtime_best );
    }

    diff = compare_matrices( m, n, c, ldc, cref, ldc );

    printf( "%d %le %le \n", p, gflops / dtime_best, diff );
    fflush( stdout );

    free( a );
    free( b );
    free( c );
    free( cold );
    free( cref );
  }

  printf( "];\n" );

  exit( 0 );
}
```

Chapter 3: Baseline

File: MMult0.c

Analysis of MMult0.c - Baseline Matrix Multiplication Implementation

1. Role & Purpose

This file serves as the **reference implementation** and **performance baseline** in a matrix multiplication optimization study. Its primary purposes are:

- **Demonstrating the canonical triple-nested loop algorithm** for dense matrix multiplication (GEMM)
- **Establishing a baseline performance metric** against which optimized versions can be compared
- **Implementing the mathematical operation $\mathbf{C} = \mathbf{A} \times \mathbf{B} + \mathbf{C}$** (GEMM with accumulation)
- **Enforcing column-major storage** convention to maintain consistency with standard numerical libraries like BLAS and LAPACK

2. Technical Details

Macro Definitions for Column-Major Access

```
#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]
```

These macros implement **column-major indexing**, where: - `lda`, `ldb`, `ldc` are **leading dimensions** (typically the number of rows in physical storage) - Element access uses `(column_index × leading_dimension + row_index)` - This convention is opposite to row-major (C/C++ native) but matches FORTRAN/BLAS standards

Function Signature and Parameters

```
void MY_MMult(int m, int n, int k, double *a, int lda,
              double *b, int ldb, double *c, int ldc)
```

- `m`: Number of rows in matrices A and C
- `n`: Number of columns in matrices B and C

- `k`: Number of columns in A / rows in B (common dimension)
- `lda, ldb, ldc`: Leading dimensions allowing for submatrix operations
- All matrices use **double-precision floating-point** arithmetic

Algorithm Implementation - Triple-Nested Loops

```
for (i=0; i<m; i++) {           // Loop over rows of C/A
  for (j=0; j<n; j++) {         // Loop over columns of C/B
    for (p=0; p<k; p++) {       // Inner product dimension
      C(i,j) += A(i,p) * B(p,j);
    }
  }
}
```

This implements the **ijk loop ordering** (row-major C, column-major inner product): - **Outer loop**: Traverses rows of output matrix C - **Middle loop**: Traverses columns of output matrix C
- **Inner loop**: Computes dot product of row i of A and column j of B - **Operation count**: Performs 2 × m × n × k floating-point operations

Memory Access Patterns

- **Matrix A**: Accessed by row (i) then column (p) → **stride-1 access** in inner loop
- **Matrix B**: Accessed by row (p) then column (j) → **stride-ldb access** in inner loop

- **Matrix C**: Each element updated k times → **high write frequency**

3. HPC Context

Performance Characteristics

1. **Computational Intensity**:
   - Theoretical FLOPs: 2mnk operations
   - Memory accesses: (m×k + k×n + m×n) × 8 bytes (double precision)
   - **Low arithmetic intensity** → memory-bound for most problem sizes
2. **Cache Behavior Issues**:
   - **Poor temporal locality**: Each element of B accessed m times, A accessed n times
   - **Poor spatial locality for B**: Inner loop accesses elements with stride `ldb`
   - **No data reuse**: Minimal cache line utilization for B and C accesses
3. **Memory Hierarchy Inefficiencies**:
   - **Register underutilization**: Only one accumulation register per output element
   - **No prefetching**: Predictable access patterns but no explicit prefetch directives
   - **Memory bandwidth saturation**: Limited by main memory bandwidth for large matrices

Optimization Opportunities

This naive implementation serves as a starting point for several key HPC optimizations:

1. **Loop Reordering**: Changing to ikj or jik ordering to improve locality
2. **Blocking/Tiling**: Dividing computation into cache-friendly submatrices
3. **Vectorization**: Using SIMD instructions (AVX, AVX-512) for parallel computation
4. **Loop Unrolling**: Reducing loop overhead and enabling instruction-level parallelism
5. **Register Tiling**: Accumulating multiple results in registers before writing to memory
6. **Parallelization**: Multi-threading (OpenMP) or distributed memory (MPI) approaches

Educational Value

- **Clear separation** of algorithmic correctness from performance optimization
- **Demonstrates fundamental limitations** of naive implementations
- **Provides measurable baseline** for quantifying optimization benefits

- **Illustrates importance** of memory access patterns over pure operation count

This implementation represents the **starting point** in the optimization journey, where subsequent versions will systematically address the performance bottlenecks through increasingly sophisticated techniques while maintaining mathematical correctness.

Source Code Implementation

```
/* Create macros so that the matrices are stored in column-major order */

#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Routine for computing C = A * B + C */

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j, p;

  for ( i=0; i<m; i++ ){        /* Loop over the rows of C */
    for ( j=0; j<n; j++ ){        /* Loop over the columns of C */
      for ( p=0; p<k; p++ ){        /* Update C( i,j ) with the inner
                        product of the ith row of A and
                        the jth column of B */
    C( i,j ) = C( i,j ) +  A( i,p ) * B( p,j );
      }
    }
  }
}
```

File: MMult1.c

Analysis of `MMult1.c`

1. Role & Purpose

This file implements **baseline matrix multiplication** (GEMM: GEneral Matrix Multiply) for a high-performance computing project infrastructure. Its

primary purposes are: - **Reference implementation**: Provides a correct, unoptimized version of $C = A \cdot B + C$ for validating optimized variants. - **Teaching tool**: Demonstrates fundamental concepts like **column-major storage** and naive triple-loop computation. - **Performance baseline**: Serves as the initial benchmark (1x speed) against which all optimized implementations are measured.

2. Technical Details

Storage Layout & Macros

```
#define A(i,j) a[ (j)*lda + (i) ]
```

- **Column-major ordering**: Element (i,j) is stored at offset `j*lda + i`, where consecutive elements in a **column** occupy contiguous memory.
- **Leading dimension (lda)**: Accounts for possible **padding** or submatrix selection, allowing the code to work with non-square memory layouts.

Matrix Multiplication Algorithm

```
for ( j=0; j<n; j+=1 ) {        // Columns of C
  for ( i=0; i<m; i+=1 ) {      // Rows of C
    AddDot(k, &A(i,0), lda, &B(0,j), &C(i,j));
  }
}
```

- **Triple-nested loop**: The outer loops (i,j) iterate over all **C(i,j)** elements.
- **Computational pattern**: Each C(i,j) accumulates the **dot product** of row i of A with column j of B.
- **AddDot function**: Computes the inner product with explicit vector indexing.

Dot Product Implementation

```
void AddDot(int k, double *x, int incx, double *y, double *gamma) {
  for (p=0; p<k; p++) {
    *gamma += X(p) * y[p];
  }
}
```

- **Strided access**: The vector `x` (a row of A) uses stride `incx = lda` due to column-major storage.
- **Unit stride access**: The vector `y` (a column of B) has implicit stride 1, enabling contiguous memory access.
- **Pointer arithmetic**: Direct pointer increments are avoided; instead, explicit indexing through macros is used.

Memory Access Patterns

- **A**: Accessed with **stride lda** (non-contiguous, poor spatial locality).
- **B**: Accessed **contiguously** within each column (good spatial locality).
- **C**: Accessed in **column-major order** (good spatial locality for this access pattern).

3. HPC Context

Performance Characteristics

- **Computational intensity**: Approximately 2k flops per C element, with O(mnk) total operations and O(mk + kn + mn) data movement.
- **Memory bottleneck**: The naive triple-loop structure results in **inefficient cache utilization**:
  - Each element of A is loaded **n times** (once per column of B).
  - Each element of B is loaded **m times** (once per row of A).
  - This leads to **O(mnk) memory accesses** versus O(mk + kn + mn) minimal possible.

Relevance to Optimization

1. **Baseline for comparison**: All optimizations (blocking, vectorization, parallelization) start from this reference.
2. **Demonstrates fundamental issues**:
   - **Poor data reuse**: No temporal locality for A or B.
   - **Inefficient ordering**: The loop structure doesn't exploit cache hierarchy.
   - **Scalar computation**: No **SIMD vectorization** or instruction-level parallelism.
3. **Column-major rationale**: Many scientific computing libraries (BLAS, LAPACK) use column-major storage for historical and mathematical consistency reasons.

Optimization Pathways

- **Loop reordering**: Changing the loop nest order can improve cache behavior.
- **Blocking/tiling**: Partition matrices to fit in cache, reusing loaded blocks.
- **Vectorization**: Using **AVX intrinsics** to compute multiple operations simultaneously.
- **Register blocking**: Unrolling loops to keep intermediate results in CPU registers.
- **Parallelization**: Distributing work across multiple cores/threads.

Key Performance Limiting Factors

- **Strided memory access** for A causes cache thrashing.

- **No prefetching** due to unpredictable access patterns.
- **Scalar operations** underutilize modern CPU vector units.
- **High cache miss rates** from repeated loading of matrix elements.

This implementation represents the starting point of the **optimization journey**, where subsequent versions will systematically address these limitations through algorithmic transformations and hardware-aware programming.

Source Code Implementation

```
/* Create macros so that the matrices are stored in column-major order */

#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Routine for computing C = A * B + C */

void AddDot( int, double *, int, double *, double * );

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j;

  for ( j=0; j<n; j+=1 ){        /* Loop over the columns of C */
    for ( i=0; i<m; i+=1 ){        /* Loop over the rows of C */
      /* Update the C( i,j ) with the inner product of the ith row of A
     and the jth column of B */

      AddDot( k, &A( i,0 ), lda, &B( 0,j ), &C( i,j ) );
    }
  }
}


/* Create macro to let X( i ) equal the ith element of x */

#define X(i) x[ (i)*incx ]

void AddDot( int k, double *x, int incx,  double *y, double *gamma )
{
  /* compute gamma := x' * y + gamma with vectors x and y of length n.

     Here x starts at location x with increment (stride) incx and y starts at location y and
```

```
  */

  int p;

  for ( p=0; p<k; p++ ){
    *gamma += X( p ) * y[ p ];
  }
}
```

File: MMult1.c

Analysis of `MMult1.c`

1. Role & Purpose

This file implements **baseline matrix multiplication** (GEMM: GEneral Matrix Multiply) for a high-performance computing project infrastructure. Its primary purposes are: - **Reference implementation**: Provides a correct, unoptimized version of $C = A \cdot B + C$ for validating optimized variants. - **Teaching tool**: Demonstrates fundamental concepts like **column-major storage** and naive triple-loop computation. - **Performance baseline**: Serves as the initial benchmark (1x speed) against which all optimized implementations are measured.

2. Technical Details

Storage Layout & Macros

```
#define A(i,j) a[ (j)*lda + (i) ]
```

- **Column-major ordering**: Element (i,j) is stored at offset `j*lda + i`, where consecutive elements in a **column** occupy contiguous memory.
- **Leading dimension (lda)**: Accounts for possible **padding** or submatrix selection, allowing the code to work with non-square memory layouts.

Matrix Multiplication Algorithm

```
for ( j=0; j<n; j+=1 ) {        // Columns of C
  for ( i=0; i<m; i+=1 ) {      // Rows of C
    AddDot(k, &A(i,0), lda, &B(0,j), &C(i,j));
  }
}
```

- **Triple-nested loop**: The outer loops (i,j) iterate over all **C(i,j)** elements.
- **Computational pattern**: Each C(i,j) accumulates the **dot product** of row i of A with column j of B.

- **AddDot function**: Computes the inner product with explicit vector indexing.

Dot Product Implementation

```
void AddDot(int k, double *x, int incx, double *y, double *gamma) {
  for (p=0; p<k; p++) {
    *gamma += X(p) * y[p];
  }
}
```

- **Strided access**: The vector `x` (a row of A) uses stride `incx = lda` due to column-major storage.
- **Unit stride access**: The vector `y` (a column of B) has implicit stride 1, enabling contiguous memory access.
- **Pointer arithmetic**: Direct pointer increments are avoided; instead, explicit indexing through macros is used.

Memory Access Patterns

- **A**: Accessed with **stride lda** (non-contiguous, poor spatial locality).
- **B**: Accessed **contiguously** within each column (good spatial locality).
- **C**: Accessed in **column-major order** (good spatial locality for this access pattern).

3. HPC Context

Performance Characteristics

- **Computational intensity**: Approximately 2k flops per C element, with O(mnk) total operations and O(mk + kn + mn) data movement.
- **Memory bottleneck**: The naive triple-loop structure results in **inefficient cache utilization**:
  - Each element of A is loaded **n times** (once per column of B).
  - Each element of B is loaded **m times** (once per row of A).
  - This leads to **O(mnk) memory accesses** versus O(mk + kn + mn) minimal possible.

Relevance to Optimization

1. **Baseline for comparison**: All optimizations (blocking, vectorization, parallelization) start from this reference.
2. **Demonstrates fundamental issues**:
   - **Poor data reuse**: No temporal locality for A or B.
   - **Inefficient ordering**: The loop structure doesn't exploit cache hierarchy.
   - **Scalar computation**: No **SIMD vectorization** or instruction-level parallelism.

3. **Column-major rationale**: Many scientific computing libraries (BLAS, LAPACK) use column-major storage for historical and mathematical consistency reasons.

Optimization Pathways

- **Loop reordering**: Changing the loop nest order can improve cache behavior.
- **Blocking/tiling**: Partition matrices to fit in cache, reusing loaded blocks.
- **Vectorization**: Using **AVX intrinsics** to compute multiple operations simultaneously.
- **Register blocking**: Unrolling loops to keep intermediate results in CPU registers.
- **Parallelization**: Distributing work across multiple cores/threads.

Key Performance Limiting Factors

- **Strided memory access** for A causes cache thrashing.
- **No prefetching** due to unpredictable access patterns.
- **Scalar operations** underutilize modern CPU vector units.
- **High cache miss rates** from repeated loading of matrix elements.

This implementation represents the starting point of the **optimization journey**, where subsequent versions will systematically address these limitations through algorithmic transformations and hardware-aware programming.

Source Code Implementation

```
/* Create macros so that the matrices are stored in column-major order */

#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Routine for computing C = A * B + C */

void AddDot( int, double *, int, double *, double * );

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j;

  for ( j=0; j<n; j+=1 ){        /* Loop over the columns of C */
    for ( i=0; i<m; i+=1 ){        /* Loop over the rows of C */
      /* Update the C( i,j ) with the inner product of the ith row of A
```

```
      and the jth column of B */

       AddDot( k, &A( i,0 ), lda, &B( 0,j ), &C( i,j ) );
    }
  }
}


/* Create macro to let X( i ) equal the ith element of x */

#define X(i) x[ (i)*incx ]

void AddDot( int k, double *x, int incx,  double *y, double *gamma )
{
  /* compute gamma := x' * y + gamma with vectors x and y of length n.

     Here x starts at location x with increment (stride) incx and y starts at location y an
  */

  int p;

  for ( p=0; p<k; p++ ){
    *gamma += X( p ) * y[ p ];
  }
}
```

File: MMult2.c

Analysis of `MMult2.c`

1. Role & Purpose

This file implements a **matrix multiplication kernel** (`C = A × B + C`) optimized through **loop unrolling**, serving as a fundamental building block in scientific computing and HPC applications. Within the optimization hierarchy, it represents **Level 2** (register-level optimization) where the primary goal is to **reduce loop overhead** and improve instruction scheduling by processing multiple output elements per iteration.

The code's infrastructure role is to provide a **performance baseline** for comparing optimization techniques, demonstrating: - **Column-major storage** consistent with BLAS/LAPACK conventions - **Loop transformation** strategies - **Register reuse** patterns - The transition from naive triple-loop implementations toward more sophisticated blocking approaches

2. Technical Details

Memory Layout & Access Patterns

- **Column-major ordering**: Implemented via macros `A(i,j)`, `B(i,j)`, `C(i,j)` where element access follows `a[(j)*lda + (i)]`
  - Matrices stored as 1D arrays with column elements contiguous in memory
  - **Leading dimensions** (`lda`, `ldb`, `ldc`) specify the stride between columns
- **Pointer arithmetic**: `&A(i,0)` computes the address of the i-th row element in the first column
- **Implicit vector stride**: In `AddDot`, vector `x` (a row of A) uses stride `incx = lda`, while vector `y` (a column of B) uses unit stride (contiguous access)

Algorithmic Structure

```
Outer loop: j (columns of C) unrolled by 4
  Middle loop: i (rows of C) stride 1
    Inner computation: 4 calls to AddDot for C(i,j:j+3)
```

Key Implementation Features

- **J-loop unrolling**: Processes 4 columns of C simultaneously
  - Reduces loop control overhead by 75%
  - Enables **instruction-level parallelism** through independent dot products
  - Improves **register reuse** of row vector from A
- **AddDot function**: Computes single dot product `+= x ·y`
  - Takes vector `x` (row of A) with stride `lda`
  - Takes vector `y` (column of B) with unit stride
  - Accumulates directly into `gamma` (element of C)
  - Uses macro `X(p)` for stride-aware access: `x[(p)*incx]`

Performance Characteristics

- **Computational intensity**: ~2k FLOPs per k elements loaded (neglecting C accesses)
- **Memory access pattern**:
  - A accessed by row: **non-contiguous** (stride `lda`)
  - B accessed by column: **contiguous** (ideal for prefetching)
  - C accessed by column: **strided** (4 elements at different offsets)
- **Register pressure**: Moderate - must hold 4 accumulators plus elements of x and y

3. HPC Context

Why This Optimization Matters

1. **Loop Overhead Reduction**:
   - Unrolling decreases branch misprediction penalties
   - Reduces loop counter maintenance operations
   - Enables better **pipeline utilization** by providing more independent operations
2. **Memory Hierarchy Optimization**:
   - **Spatial locality**: Processing 4 adjacent columns of C improves cache line utilization for B (contiguous access)
   - **Temporal locality**: Row vector from A reused across 4 dot products, reducing memory traffic
   - **Register blocking**: The 4 accumulators (`C(i,j:j+3)`) can be kept in registers
3. **Instruction-Level Parallelism (ILP)**:
   - Multiple independent dot products allow out-of-order execution
   - Compiler can schedule loads and FMAs more efficiently
   - Reduces data dependency stalls

Performance Limitations & Trade-offs

1. **Suboptimal Cache Utilization**:
   - No **blocking/tiling** for L1/L2 cache
   - Row access of A causes **cache thrashing** for large matrices
   - Only exploits register reuse, not cache reuse
2. **Vectorization Challenges**:
   - Inner `AddDot` loop remains scalar
   - Compiler cannot easily vectorize due to strided access in A
   - Missed opportunity for SIMD parallelism
3. **Arithmetic Intensity**:
   - Still O(k) memory operations per 2k FLOPs
   - Limited by loading entire row of A for each i iteration

Educational Value in Optimization Progression

This code represents **Step 2** in the optimization journey: 1. **Baseline**: Triple nested loops (ijk order) 2. **This version**: Loop unrolling + dot product kernel 3. **Next steps**: Register blocking → cache blocking → vectorization → parallelization

Modern HPC Relevance

- Demonstrates fundamental optimization principles still applicable to GPU kernels
- Illustrates the **optimization space** between algorithm and hardware

- Shows the importance of **data layout** choices (column-major vs row-major)
- Provides foundation for understanding **BLAS-level 3** implementations

Key Performance Takeaways

- **Unrolling factor 4** balances register usage and ILP for typical architectures
- **Column-major** storage benefits contiguous access patterns in B
- The **AddDot abstraction** separates optimization concerns (inner loop vs outer loops)
- This optimization typically provides **2-3× speedup** over naive implementation but remains far from peak hardware performance

This implementation serves as a crucial teaching example in HPC education, illustrating how **micro-architectural awareness** drives algorithmic transformations, setting the stage for more advanced techniques like SIMD vectorization and cache-aware blocking.

Source Code Implementation

```
/* Create macros so that the matrices are stored in column-major order */

#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Routine for computing C = A * B + C */

void AddDot( int, double *, int, double *, double * );

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j;

  for ( j=0; j<n; j+=4 ){        /* Loop over the columns of C, unrolled by 4 */
    for ( i=0; i<m; i+=1 ){        /* Loop over the rows of C */
      /* Update the C( i,j ) with the inner product of the ith row of A
     and the jth column of B */

      AddDot( k, &A( i,0 ), lda, &B( 0,j ), &C( i,j ) );

      /* Update the C( i,j+1 ) with the inner product of the ith row of A
     and the (j+1)th column of B */
```

```
      AddDot( k, &A( i,0 ), lda, &B( 0,j+1 ), &C( i,j+1 ) );

      /* Update the C( i,j+2 ) with the inner product of the ith row of A
     and the (j+2)th column of B */

      AddDot( k, &A( i,0 ), lda, &B( 0,j+2 ), &C( i,j+2 ) );

      /* Update the C( i,j+3 ) with the inner product of the ith row of A
     and the (j+1)th column of B */

      AddDot( k, &A( i,0 ), lda, &B( 0,j+3 ), &C( i,j+3 ) );
    }
  }
}


/* Create macro to let X( i ) equal the ith element of x */

#define X(i) x[ (i)*incx ]

void AddDot( int k, double *x, int incx,  double *y, double *gamma )
{
  /* compute gamma := x' * y + gamma with vectors x and y of length n.

     Here x starts at location x with increment (stride) incx and y starts at location y and
  */

  int p;

  for ( p=0; p<k; p++ ){
    *gamma += X( p ) * y[ p ];
  }
}
```

Chapter 4: Scalar Opt (1x4)

File: MMult_1x4_3.c

Technical Analysis of MMult_1x4_3.c

1. Role & Purpose in Project Infrastructure

This file implements a **matrix multiplication kernel** within a progressive optimization framework for High Performance Computing. It represents **Stage 3** in a series of optimization steps, specifically demonstrating **loop unrolling across four columns** of the output matrix C. The primary objective is to **reduce loop overhead** and **improve instruction-level parallelism** while maintaining the correct matrix multiplication algorithm C = A × B + C (GEMM operation). This code serves as an educational stepping stone between naive implementations and more advanced optimizations like vectorization and cache blocking.

2. Technical Details

Matrix Storage and Access Macros

```
#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]
```

- These macros enforce **column-major storage** (Fortran-style) where consecutive elements in memory belong to the same column
- The `lda`, `ldb`, `ldc` parameters are **leading dimensions** allowing access to sub-matrices within larger allocated blocks
- The indexing `(j)*ld + (i)` calculates the offset: column index × row stride + row index

Main Multiplication Kernel

```
void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j;
  for ( j=0; j<n; j+=4 ){        /* Loop unrolled by 4 in j-dimension */
    for ( i=0; i<m; i+=1 ){        /* Process one row at a time */
      AddDot1x4( k, &A( i,0 ), lda, &B( 0,j ), ldb, &C( i,j ), ldc );
    }
  }
}
```

- **Loop structure**: Outer loop on columns (j) unrolled by 4, inner loop on rows (i)
- **Memory access pattern**: Processes one row of A against four columns of B simultaneously
- The `AddDot1x4` call computes **four output elements** C(i,j:j+3) using one row of A and four columns of B

Four-Element Computation Routine

```
void AddDot1x4( int k, double *a, int lda, double *b, int ldb,
                double *c, int ldc )
{
  AddDot( k, &A( 0, 0 ), lda, &B( 0, 0 ), &C( 0, 0 ) );
  AddDot( k, &A( 0, 0 ), lda, &B( 0, 1 ), &C( 0, 1 ) );
  AddDot( k, &A( 0, 0 ), lda, &B( 0, 2 ), &C( 0, 2 ) );
  AddDot( k, &A( 0, 0 ), lda, &B( 0, 3 ), &C( 0, 3 ) );
}
```

- **Key optimization**: Four independent dot products share the same row of A
- Each call computes C(i,j+p) = Σ A(i,:) × B(:,j+p) for p = 0..3
- The routine exploits **data reuse** of A elements across multiple dot products

Scalar Dot Product Implementation

```
void AddDot( int k, double *x, int incx, double *y, double *gamma )
{
  for ( int p=0; p<k; p++ ){
    *gamma += X( p ) * y[ p ];  /* Scalar accumulation */
  }
}
```

- **Strided access**: `X(p)` macro handles `incx` stride for row elements
- **Column-major implication**: B columns accessed with unit stride (contiguous in memory)
- **Critical path**: Serial dependency through `*gamma` accumulation limits parallelism

3. HPC Context and Performance Implications

Loop Unrolling Benefits

- **Reduced loop overhead**: The j-loop runs n/4 iterations instead of n, decreasing branch prediction misses
- **Instruction scheduling**: Four independent dot products allow better pipeline utilization

- **Register reuse**: Compiler can keep A(i,p) in a register across the four dot products

Memory Access Patterns

- **A matrix**: Accessed with stride `lda` (non-unit stride) - suboptimal for cache lines
- **B matrix**: Four columns accessed with unit stride - optimal for spatial locality
- **C matrix**: Four elements written with stride `ldc` - poor spatial locality for column-major

Limitations and Optimization Opportunities

1. **No vectorization**: Scalar operations don't utilize SIMD (AVX/SSE) instructions
2. **No cache blocking**: Entire matrices may not fit in cache for large problems
3. **Register pressure**: Four accumulators might be insufficient for modern wide SIMD
4. **Loop order**: j-i-p loop nest isn't optimal for cache hierarchy

Performance Characteristics

- **Arithmetic Intensity**: Approximately 2k/(k+4) FLOPs per byte (assuming 8-byte doubles)
- **Potential speedup**: ~2-3x over naive triple loop from reduced loop overhead
- **Bottleneck**: Memory bandwidth for large matrices due to lack of cache optimization

Educational Value

This implementation demonstrates **fundamental optimization principles**: 1. **Loop unrolling** to reduce overhead 2. **Data reuse** across multiple operations 3. **Stride-aware access patterns** 4. **Modular design** for progressive optimization

The code serves as a foundation for more advanced techniques like **register tiling** (to increase unrolling factor), **vectorization** (using AVX intrinsics), and **cache blocking** (for multi-level memory hierarchy optimization).

Source Code Implementation

```
/* Create macros so that the matrices are stored in column-major order */

#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
```

```
#define C(i,j) c[ (j)*ldc + (i) ]

/* Routine for computing C = A * B + C */

void AddDot( int, double *, int, double *, double * );
void AddDot1x4( int, double *, int,  double *, int, double *, int );

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j;

  for ( j=0; j<n; j+=4 ){        /* Loop over the columns of C, unrolled by 4 */
    for ( i=0; i<m; i+=1 ){        /* Loop over the rows of C */
      /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
     one routine (four inner products) */

      AddDot1x4( k, &A( i,0 ), lda, &B( 0,j ), ldb, &C( i,j ), ldc );
    }
  }
}


void AddDot1x4( int k, double *a, int lda,  double *b, int ldb, double *c, int ldc )
{
  /* So, this routine computes four elements of C:

           C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ).

     Notice that this routine is called with c = C( i, j ) in the
     previous routine, so these are actually the elements

           C( i, j ), C( i, j+1 ), C( i, j+2 ), C( i, j+3 )

     in the original matrix C */

  AddDot( k, &A( 0, 0 ), lda, &B( 0, 0 ), &C( 0, 0 ) );
  AddDot( k, &A( 0, 0 ), lda, &B( 0, 1 ), &C( 0, 1 ) );
  AddDot( k, &A( 0, 0 ), lda, &B( 0, 2 ), &C( 0, 2 ) );
  AddDot( k, &A( 0, 0 ), lda, &B( 0, 3 ), &C( 0, 3 ) );
}


/* Create macro to let X( i ) equal the ith element of x */
```

```c
#define X(i) x[ (i)*incx ]

void AddDot( int k, double *x, int incx,  double *y, double *gamma )
{
  /* compute gamma := x' * y + gamma with vectors x and y of length n.

     Here x starts at location x with increment (stride) incx and y starts at location y and

  */

  int p;

  for ( p=0; p<k; p++ ){
    *gamma += X( p ) * y[ p ];
  }
}
```

File: MMult_1x4_4.c

Analysis of `MMult_1x4_4.c`

1. Role & Purpose

This file implements a **matrix multiplication kernel** for computing **C = A × B + C** within a larger optimization framework. It serves as an **intermediate optimization step** in a progression from naive triple-loop implementations toward highly-tuned HPC routines. Specifically, it demonstrates: - **Loop unrolling** in the column dimension of C/B (by a factor of 4) - **Manual inlining** of dot product computations - A **column-major storage** convention consistent with BLAS/LAPACK standards

This version is designed to **reduce loop overhead** and **improve instruction-level parallelism** while maintaining correctness, forming a foundation for more advanced optimizations like vectorization and cache blocking.

2. Technical Details

Storage and Access Macros:

```c
#define A(i,j) a[ (j)*lda + (i) ]  // Column-major indexing
```

- Matrices are stored in **column-major order**: consecutive rows of a column occupy adjacent memory locations.
- `lda` (leading dimension of A) typically equals the row count `m`, enabling proper striding through memory.
- These macros abstract the indexing complexity while ensuring correct memory access patterns.

Main Routine `MY_MMult`: - **Loop structure**: Outer loop steps through columns of C in increments of 4 (`j+=4`), inner loop processes rows one at a time (`i+=1`). - **Function call**: For each (i,j) pair, calls `AddDot1x4` to compute four output elements: `C(i,j), C(i,j+1), C(i,j+2), C(i,j+3)` - **Parameter passing**: - `&A(i,0)`: Points to the beginning of row `i` of A (stride `lda` between columns) - `&B(0,j)`: Points to the beginning of column `j` of B (stride `ldb` between rows) - `&C(i,j)`: Points to the first of four consecutive column elements in row `i`

Kernel Routine `AddDot1x4`: - Computes four independent dot products for a **single row** of A against **four columns** of B. - **Explicit unrolling**: Four separate inner loops (over `p`) compute: `C(0,0) += Σ A(0,p) * B(p,0) // p=0..k-1  C(0,1) += Σ A(0,p) * B(p,1)  C(0,2) += Σ A(0,p) * B(p,2)  C(0,3) += Σ A(0,p) * B(p,3)` - **Memory access pattern**: - A: Row-wise access with stride `lda` (non-contiguous in column-major storage) - B: Accesses four columns simultaneously with stride `ldb` between rows - C: Updates four consecutive column elements in the same row

3. HPC Context

Performance Characteristics: 1. **Loop Overhead Reduction**: - Unrolling the column loop reduces branch instructions by a factor of 4. - Enables better instruction scheduling and reduces pipeline stalls.

2. **Data Reuse Opportunities**:
   - Each element `A(0,p)` is reused across four multiplications (once per column of B).
   - This **register-level reuse** is explicit but not yet optimized—the four loops load `A(0,p)` separately.
3. **Memory Access Patterns**:
   - **A access**: Strided by `lda` (poor spatial locality in column-major).
   - **B access**: Four columns accessed simultaneously, but each column access is contiguous (good for prefetching).
   - **C access**: Four contiguous writes in row `i` (excellent spatial locality).
4. **Foundation for Vectorization**:
   - The four independent dot products are **prime candidates for SIMD instructions**.
   - Future optimizations could:
     - Load four `B(p,*)` values into a SIMD register
     - Broadcast `A(0,p)` and perform a fused multiply-add
     - Use AVX/SSE intrinsics for 256-bit/128-bit operations
5. **Limitations and Next Steps**:
   - **No cache blocking**: Still suffers from poor cache utilization for large matrices.
   - **No actual SIMD instructions**: The unrolled structure is ready for vectorization but doesn't implement it.
   - **Inefficient A access**: Row-wise access in column-major order causes

cache thrashing.

- **Manual inlining** is beneficial but should be complemented with compiler optimizations (`-funroll-loops`, `-ffast-math`).

Educational Significance: This code exemplifies a **critical transitional optimization** in the matrix multiplication optimization hierarchy: - Step 1: Naive triple loop - Step 2: Loop unrolling (this version) - Step 3: Register tiling + vectorization - Step 4: Cache blocking + SIMD - Step 5: Multi-threading + NUMA optimization

It clearly demonstrates how **exposing parallelism** through loop unrolling enables subsequent vectorization, making it a foundational pattern in HPC kernel development. The explicit column-major storage also highlights the importance of **memory layout awareness** when optimizing for modern cache hierarchies.

Source Code Implementation

```
/* Create macros so that the matrices are stored in column-major order */

#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Routine for computing C = A * B + C */

void AddDot1x4( int, double *, int,  double *, int, double *, int );

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j;

  for ( j=0; j<n; j+=4 ){        /* Loop over the columns of C, unrolled by 4 */
    for ( i=0; i<m; i+=1 ){        /* Loop over the rows of C */
      /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
     one routine (four inner products) */

      AddDot1x4( k, &A( i,0 ), lda, &B( 0,j ), ldb, &C( i,j ), ldc );
    }
  }
}


void AddDot1x4( int k, double *a, int lda,  double *b, int ldb, double *c, int ldc )
{
  /* So, this routine computes four elements of C:
```

```
           C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ).

       Notice that this routine is called with c = C( i, j ) in the
       previous routine, so these are actually the elements

           C( i, j ), C( i, j+1 ), C( i, j+2 ), C( i, j+3 )

       in the original matrix C.

       In this version, we "inline" AddDot */

  int p;

  //  AddDot( k, &A( 0, 0 ), lda, &B( 0, 0 ), &C( 0, 0 ) );
  for ( p=0; p<k; p++ ){
    C( 0, 0 ) += A( 0, p ) * B( p, 0 );
  }

  //  AddDot( k, &A( 0, 0 ), lda, &B( 0, 1 ), &C( 0, 1 ) );
  for ( p=0; p<k; p++ ){
    C( 0, 1 ) += A( 0, p ) * B( p, 1 );
  }

  //  AddDot( k, &A( 0, 0 ), lda, &B( 0, 2 ), &C( 0, 2 ) );
  for ( p=0; p<k; p++ ){
    C( 0, 2 ) += A( 0, p ) * B( p, 2 );
  }

  //  AddDot( k, &A( 0, 0 ), lda, &B( 0, 3 ), &C( 0, 3 ) );
  for ( p=0; p<k; p++ ){
    C( 0, 3 ) += A( 0, p ) * B( p, 3 );
  }
}
```

File: MMult_1x4_5.c

Analysis of `MMult_1x4_5.c`

1. Role & Purpose

This file implements **matrix multiplication with column-major storage and loop unrolling** within a broader HPC optimization study. Specifically, it demonstrates **1×4 kernel unrolling** where the inner product computation for four consecutive columns of matrix C is **merged into a single loop**. This

serves as an intermediate optimization step between naive triple-nested loops and more advanced techniques like SIMD vectorization, playing a **foundational role** in teaching the progressive optimization of compute-bound kernels.

2. Technical Details

Macro Definitions: - The macros `A(i,j)`, `B(i,j)`, and `C(i,j)` implement **column-major storage** where element access is `a[(j)*lda + (i)]`. This matches Fortran/LAPACK conventions and affects memory access patterns crucial for cache performance.

Main Function `MY_MMult`: - **Loop structure**: Outer loop over columns of C (`j`) with **unrolling factor 4** (step size 4), inner loop over rows of C (`i`) with step 1. - **Kernel invocation**: Calls `AddDot1x4` for each (i, j) block, passing: - `k`: Inner dimension of matrix multiplication (common dimension of A and B) - `&A(i,0)`: Pointer to the ith row of A (starting at column 0) - `&B(0,j)`: Pointer to the jth column of B (starting at row 0) - `&C(i,j)`: Pointer to the C(i,j) element where results accumulate

Kernel Function `AddDot1x4`: - **Functionality**: Computes four inner products simultaneously: `C(i,j) += Σ A(i,p)*B(p,j) C(i,j+1) += Σ A(i,p)*B(p,j+1) C(i,j+2) += Σ A(i,p)*B(p,j+2) C(i,j+3) += Σ A(i,p)*B(p,j+3)` - **Loop fusion**: The four separate inner product loops are merged into one loop over `p`, reducing loop overhead and improving **data locality**. - **Memory access pattern**: - `A(0,p)`: Accesses consecutive memory locations in the same row of A (stride 1) - `B(p,0..3)`: Accesses four consecutive elements in the same row of B (stride 1) - This pattern enables potential **spatial locality** benefits in cache

Key Implementation Notes: - No explicit SIMD intrinsics are used; the optimization is purely at the **C loop level** - The **register blocking** is implicit: four C elements are accumulated simultaneously, likely keeping them in registers - The kernel computes a **1×4 micro-panel** of matrix C

3. HPC Context

Performance Relevance:

1. **Loop Unrolling Benefits**:
   - **Reduced loop overhead**: The inner `p` loop executes `k` iterations instead of `4×k` if separate loops were used
   - **Improved instruction scheduling**: The compiler has more operations to schedule, potentially hiding **pipeline latencies**
   - **Register reuse**: The same `A(0,p)` value is used for four multiplications, reducing register pressure
2. **Memory Access Optimization**:
   - **Spatial locality**: Accessing four consecutive `B` elements exploits cache line utilization (typically 64 bytes)

- **Reduced pointer arithmetic**: Computing offsets once per iteration for all four products
- **Column-major consideration**: While B accesses are contiguous, A accesses within a column would have stride `lda`; here A accesses are along rows due to the specific kernel structure

3. **Foundation for Advanced Optimizations**:
   - This **1×4 unrolling** prepares for **SIMD vectorization** (e.g., using AVX2 to process four doubles in one instruction)
   - Demonstrates the **kernel fusion** principle critical for register blocking
   - Serves as a stepping stone to **cache-aware blocking** where larger blocks of A, B, and C are staged in cache

Limitations and Next Steps: - Still suffers from **cache misses** for larger matrices as no explicit cache blocking is implemented - Only unrolls the column dimension; further unrolling of both row and column dimensions could improve performance - No explicit **vectorization**; relies on compiler auto-vectorization which may be suboptimal - Fixed unrolling factor (4) may not match the system's optimal **vector width**

Educational Value: This code exemplifies the **progressive optimization methodology** in HPC: starting from naive loops, introducing unrolling, then proceeding to vectorization and cache blocking. It clearly demonstrates how **loop transformations** can improve performance even without hardware-specific intrinsics, making it an excellent teaching example for **architecture-aware programming**.

Source Code Implementation

```
/* Create macros so that the matrices are stored in column-major order */

#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Routine for computing C = A * B + C */

void AddDot1x4( int, double *, int,  double *, int, double *, int );

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j;

  for ( j=0; j<n; j+=4 ){        /* Loop over the columns of C, unrolled by 4 */
    for ( i=0; i<m; i+=1 ){        /* Loop over the rows of C */
```

```
      /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
      one routine (four inner products) */

      AddDot1x4( k, &A( i,0 ), lda, &B( 0,j ), ldb, &C( i,j ), ldc );
    }
  }
}


void AddDot1x4( int k, double *a, int lda,  double *b, int ldb, double *c, int ldc )
{
  /* So, this routine computes four elements of C:

           C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ).

     Notice that this routine is called with c = C( i, j ) in the
     previous routine, so these are actually the elements

           C( i, j ), C( i, j+1 ), C( i, j+2 ), C( i, j+3 )

     in the original matrix C.

     In this version, we merge the four loops, computing four inner
     products simultaneously. */

  int p;

  //  AddDot( k, &A( 0, 0 ), lda, &B( 0, 0 ), &C( 0, 0 ) );
  //  AddDot( k, &A( 0, 0 ), lda, &B( 0, 1 ), &C( 0, 1 ) );
  //  AddDot( k, &A( 0, 0 ), lda, &B( 0, 2 ), &C( 0, 2 ) );
  //  AddDot( k, &A( 0, 0 ), lda, &B( 0, 3 ), &C( 0, 3 ) );
  for ( p=0; p<k; p++ ){
    C( 0, 0 ) += A( 0, p ) * B( p, 0 );
    C( 0, 1 ) += A( 0, p ) * B( p, 1 );
    C( 0, 2 ) += A( 0, p ) * B( p, 2 );
    C( 0, 3 ) += A( 0, p ) * B( p, 3 );
  }
}
```

File: MMult_1x4_6.c

Analysis of MMult_1x4_6.c

1. Role & Purpose

This file implements **matrix multiplication optimization** within a larger HPC learning framework exploring performance optimization techniques. Specifically, it demonstrates **loop unrolling and register reuse** for computing small blocks of the output matrix. The code computes $\mathbf{C} = \mathbf{A} \times \mathbf{B} + \mathbf{C}$ using:

- **1×4 micro-kernel**: Computes one row × four columns of output per invocation
- **Column-major storage**: Matrices stored by columns (FORTRAN-style)
- **Register optimization**: Intermediate results held in CPU registers
- **Blocked computation**: Outer loops process C in 1×4 blocks

This represents an intermediate optimization level between naive triple-loop implementation and more advanced techniques using vectorization or cache blocking.

2. Technical Details

Macro Definitions for Column-Major Ordering

```
#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]
```

These macros implement **column-major addressing**, where element *(i,j)* is at position `j*ld + i`. This contrasts with row-major order (C/C++ default) and aligns with BLAS/LAPACK conventions. The leading dimension (`lda`, `ldb`, `ldc`) allows working with submatrices.

Main Multiplication Routine

```
void MY_MMult( int m, int n, int k, double *a, int lda,
               double *b, int ldb, double *c, int ldc )
{
  for ( j=0; j<n; j+=4 ) {        /* Unroll columns by 4 */
    for ( i=0; i<m; i+=1 ) {      /* Process rows singly */
      AddDot1x4( k, &A(i,0), lda, &B(0,j), ldb, &C(i,j), ldc );
    }
  }
}
```

The outer loops traverse matrix C with **4×1 blocks**. Key characteristics: - **j-loop**: Steps by 4 (column unrolling) - **i-loop**: Steps by 1 (no row unrolling)

- **Parameter passing**: `&A(i,0)` passes pointer to row i of A, `&B(0,j)` passes pointer to column j of B

Micro-Kernel Implementation

```
void AddDot1x4( int k, double *a, int lda, double *b, int ldb,
                double *c, int ldc )
{
  int p;
  register double c_00_reg, c_01_reg, c_02_reg, c_03_reg, a_0p_reg;

  c_00_reg = 0.0; c_01_reg = 0.0; c_02_reg = 0.0; c_03_reg = 0.0;

  for ( p=0; p<k; p++ ) {
    a_0p_reg = A( 0, p );  /* Load once, use 4 times */

    c_00_reg += a_0p_reg * B( p, 0 );
    c_01_reg += a_0p_reg * B( p, 1 );
    c_02_reg += a_0p_reg * B( p, 2 );
    c_03_reg += a_0p_reg * B( p, 3 );
  }

  C( 0, 0 ) += c_00_reg;
  C( 0, 1 ) += c_01_reg;
  C( 0, 2 ) += c_02_reg;
  C( 0, 3 ) += c_03_reg;
}
```

Critical optimizations: - **Register accumulation**: Four separate registers hold dot products - **Common subexpression elimination**: `a_0p_reg` loaded once, reused four times - **Loop fusion**: Single p-loop computes four dot products simultaneously - **Register keyword**: Suggests compiler keep variables in registers (though modern compilers often ignore this hint)

Memory Access Pattern

- **Matrix A**: Accessed sequentially by rows within micro-kernel (good for cache)
- **Matrix B**: Accessed with stride `ldb` across four columns (potentially cache-unfriendly)
- **Matrix C**: Single write per 4 elements after full accumulation

3. HPC Context

Performance Relevance

1. **Register Pressure Management**: The 1×4 blocking balances register usage—four accumulators plus one loaded value fit comfortably in most

CPU register files without spilling to memory.

2. **Operation Intensity**: This implementation increases **arithmetic intensity** by:
   - Loading `A(0,p)` once for four multiply-add operations (4 FLOPs/load)
   - Accumulating in registers (avoiding repeated C accesses)
   - Achieving better FLOP-to-memory ratio than naive 3-loop implementation
3. **Instruction-Level Parallelism (ILP)**: The four independent multiply-add operations within the p-loop can be **pipelined** by superscalar processors, potentially achieving up to 4-way instruction parallelism.
4. **Loop Overhead Reduction**: Unrolling the column dimension reduces loop control instructions by factor of 4 in the j-dimension.

Limitations and Evolution

- **No SIMD vectorization**: Uses scalar operations; later versions would use SSE/AVX intrinsics
- **Limited cache optimization**: No explicit cache blocking; poor spatial locality for B accesses
- **Register underutilization**: Only uses 5 registers; modern CPUs have 16-32 vector registers
- **Fixed unroll factor**: Hard-coded 1×4 blocks; not adaptable to different architectures

Performance Impact Estimate Compared to naive implementation, this optimization typically yields **2-3× speedup** on modern CPUs due to: - Reduced loop overhead - Better register reuse - Improved instruction scheduling

However, it still falls short of peak performance due to: - Suboptimal cache utilization - Lack of explicit vectorization - Limited parallelism exploitation

This code represents a **foundational optimization step** in the matrix multiplication optimization hierarchy, demonstrating the importance of **register blocking and micro-kernel design** before introducing more advanced techniques like cache blocking, SIMD, and multithreading.

Source Code Implementation

```
/* Create macros so that the matrices are stored in column-major order */

#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]
```

```
/* Routine for computing C = A * B + C */

void AddDot1x4( int, double *, int,  double *, int, double *, int )

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j;

  for ( j=0; j<n; j+=4 ){        /* Loop over the columns of C, unrolled by 4 */
    for ( i=0; i<m; i+=1 ){        /* Loop over the rows of C */
      /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
     one routine (four inner products) */

      AddDot1x4( k, &A( i,0 ), lda, &B( 0,j ), ldb, &C( i,j ), ldc );
    }
  }
}


void AddDot1x4( int k, double *a, int lda,  double *b, int ldb, double *c, int ldc )
{
  /* So, this routine computes four elements of C:

           C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ).

     Notice that this routine is called with c = C( i, j ) in the
     previous routine, so these are actually the elements

           C( i, j ), C( i, j+1 ), C( i, j+2 ), C( i, j+3 )

     in the original matrix C.

     In this version, we accumulate in registers and put A( 0, p ) in a register */

  int p;
  register double
    /* hold contributions to
       C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ) */
       c_00_reg,   c_01_reg,   c_02_reg,   c_03_reg,
    /* holds A( 0, p ) */
       a_0p_reg;

  c_00_reg = 0.0;
  c_01_reg = 0.0;
```

```
  c_02_reg = 0.0;
  c_03_reg = 0.0;

  for ( p=0; p<k; p++ ){
    a_0p_reg = A( 0, p );

    c_00_reg += a_0p_reg * B( p, 0 );
    c_01_reg += a_0p_reg * B( p, 1 );
    c_02_reg += a_0p_reg * B( p, 2 );
    c_03_reg += a_0p_reg * B( p, 3 );
  }

  C( 0, 0 ) += c_00_reg;
  C( 0, 1 ) += c_01_reg;
  C( 0, 2 ) += c_02_reg;
  C( 0, 3 ) += c_03_reg;
}
```

File: MMult_1x4_7.c

Analysis of MMult_1x4_7.c

1. Role & Purpose

This file implements a **single-row, multiple-column** kernel for matrix multiplication ($C = A \times B + C$) within a larger optimization study. It represents an **intermediate optimization stage** between naive triple-loop implementations and fully optimized BLAS-level routines. The key innovation is the **column-wise unrolling** of the computation with explicit pointer arithmetic to reduce loop overhead and improve register usage. This serves as a pedagogical example of how **manual loop unrolling and register blocking** can improve performance before introducing SIMD vectorization.

2. Technical Details

Matrix Storage & Access Patterns: - The `A(i,j)` macros implement **column-major ordering**, where element (i,j) is at position `a[j*lda + i]`. This differs from row-major (C/C++ default) where adjacent row elements are contiguous. - In column-major, columns are stored contiguously, so accessing `B(0,0), B(1,0), B(2,0)...` has **optimal spatial locality**.

Core Algorithm Structure:

```
MY_MMult (m, n, k):
  for j in [0, n) step 4:          // Process 4 columns of C/B at once
    for i in [0, m):               // Process each row of A
```

```
AddDot1x4(...)                    // Compute 4 dot products
```

AddDot1x4 Kernel Mechanics: 1. **Pointer Initialization:** `c bp0_pntr = &B(0,0); bp1_pntr = &B(0,1); ...` Four separate pointers track positions in four consecutive columns of B. This eliminates the need for computing `B(p,j)` addresses in the inner loop.

2. **Register Accumulation:**

   ```
   register double c_00_reg, c_01_reg, ...;
   ```

   The `register` keyword (now largely advisory for compilers) hints that accumulators should stay in CPU registers, reducing memory traffic.

3. **Inner Loop Computation:**

   ```
   for p in [0, k):
     a_0p_reg = A(0, p);           // Load one element from A
     c_00_reg += a_0p_reg * *bp0_pntr++;  // Multiply & accumulate with pointer auto-incre
     // ... repeat for 3 more columns
   ```

   - **Single element reuse**: One `a_0p_reg` is reused across 4 multiplications (reducing A-memory accesses per FLOP).
   - **Pointer auto-increment**: The post-increment (`*bp0_pntr++`) advances each pointer to the next row in its respective column.
   - **Column-major advantage**: Each pointer accesses contiguous memory as it increments (ideal for prefetching).

Critical Implementation Nuance: The kernel computes `C(0,j:j+3) += A(0,:) × B(:,j:j+3)` for a single row of A. The outer loops in `MY_MMult` iterate this across all rows and column blocks. This is a **1×4 micro-kernel** – it computes 1 row × 4 columns of C.

3. HPC Context

Performance Relevance:

1. **Reduced Loop Overhead:**
   - The `AddDot1x4` kernel computes 4 inner products with **one inner loop** instead of four separate loops. This amortizes loop control overhead (increment/comparison) across 4 computations.
2. **Register Blocking:**
   - The 4 accumulators (`c_xx_reg`) are explicitly kept in registers, minimizing writes to C until the final store. This leverages the **register file** – the fastest memory hierarchy level.
3. **Memory Access Patterns:**
   - **Spatial locality on B**: Each pointer traverses a column contiguously, matching the column-major storage perfectly.
   - **Temporal locality on A**: The same `a_0p_reg` is reused 4 times before loading the next A element.

- However, this kernel has **poor cache behavior for large matrices**: It streams through entire columns of B without reuse across different i iterations (no cache blocking).

4. **Pipeline Efficiency:**
   - The simple loop structure with independent accumulations allows for **instruction-level parallelism (ILP)**. Modern CPUs can execute multiple floating-point operations concurrently when dependencies are minimal.
5. **Foundation for Further Optimization:**
   - This scalar unrolled version is a prerequisite for **SIMD vectorization** (e.g., using AVX instructions). The 4 independent dot products could become a single vector operation in later versions.
   - The 1×4 shape is a stepping stone to larger **register tiles** (like 4×4 or 8×4) that better exploit cache hierarchy.

Limitations & Trade-offs: - **No cache blocking**: For matrices larger than L1 cache, performance degrades due to cache misses. - **Fixed unrolling factor**: The 1×4 shape may not be optimal for all architectures. - **Scalar computation**: Modern CPUs can perform 4-8 double-precision operations per cycle with SIMD; this code uses only scalar units. - **Register pressure**: Adding more accumulators (for larger unrolling) would compete for limited architectural registers.

Educational Value: This code demonstrates the **progressive optimization methodology** central to HPC: start with correct algorithms, then reduce overhead (loop unrolling), improve locality (blocking), and finally exploit parallelism (vectorization). The explicit pointer arithmetic and register variables make data movement patterns visible, which is crucial for understanding later SIMD and cache-blocked implementations.

Source Code Implementation

```
/* Create macros so that the matrices are stored in column-major order */

#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Routine for computing C = A * B + C */

void AddDot1x4( int, double *, int,  double *, int, double *, int )

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j;
```

```
  for ( j=0; j<n; j+=4 ){        /* Loop over the columns of C, unrolled by 4 */
    for ( i=0; i<m; i+=1 ){        /* Loop over the rows of C */
      /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
     one routine (four inner products) */

      AddDot1x4( k, &A( i,0 ), lda, &B( 0,j ), ldb, &C( i,j ), ldc );
    }
  }
}


void AddDot1x4( int k, double *a, int lda,  double *b, int ldb, double *c, int ldc )
{
  /* So, this routine computes four elements of C:

           C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ).

     Notice that this routine is called with c = C( i, j ) in the
     previous routine, so these are actually the elements

           C( i, j ), C( i, j+1 ), C( i, j+2 ), C( i, j+3 )

     in the original matrix C.

     In this version, we use pointer to track where in four columns of B we are */

  int p;
  register double
    /* hold contributions to
       C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ) */
       c_00_reg,   c_01_reg,   c_02_reg,   c_03_reg,
    /* holds A( 0, p ) */
       a_0p_reg;
  double
    /* Point to the current elements in the four columns of B */
    *bp0_pntr, *bp1_pntr, *bp2_pntr, *bp3_pntr;

  bp0_pntr = &B( 0, 0 );
  bp1_pntr = &B( 0, 1 );
  bp2_pntr = &B( 0, 2 );
  bp3_pntr = &B( 0, 3 );

  c_00_reg = 0.0;
  c_01_reg = 0.0;
  c_02_reg = 0.0;
```

```
  c_03_reg = 0.0;

  for ( p=0; p<k; p++ ){
    a_0p_reg = A( 0, p );

    c_00_reg += a_0p_reg * *bp0_pntr++;
    c_01_reg += a_0p_reg * *bp1_pntr++;
    c_02_reg += a_0p_reg * *bp2_pntr++;
    c_03_reg += a_0p_reg * *bp3_pntr++;
  }

  C( 0, 0 ) += c_00_reg;
  C( 0, 1 ) += c_01_reg;
  C( 0, 2 ) += c_02_reg;
  C( 0, 3 ) += c_03_reg;
}
```

File: MMult_1x4_8.c

Analysis of MMult_1x4_8.c

1. Role & Purpose

This file implements **matrix multiplication $\mathbf{C} = \mathbf{A} \times \mathbf{B} + \mathbf{C}$** using **manual loop unrolling and register optimization** techniques. In the broader optimization hierarchy, it represents an intermediate step that demonstrates:

- **Column-wise unrolling**: Processing multiple columns (4) of matrix C simultaneously
- **Register blocking**: Using CPU registers to accumulate multiple partial results
- **Reduced loop overhead**: Decreasing the frequency of loop control operations

This implementation serves as a **foundational building block** for more advanced optimizations like cache blocking and SIMD vectorization, showing how to exploit instruction-level parallelism before addressing memory hierarchy constraints.

2. Technical Details

Storage Order and Access Patterns

```
#define A(i,j) a[ (j)*lda + (i) ]  // Column-major indexing
```

- **Column-major storage**: Matrices are stored column-wise in memory

- **Memory layout**: Consecutive elements in a column occupy adjacent memory locations
- **Access pattern**: The inner loop traverses `A` row-wise (contiguous) and `B` column-wise (strided)

Core Algorithm Structure

```
for (j=0; j<n; j+=4) {         // Process 4 columns at a time
  for (i=0; i<m; i+=1) {       // Process 1 row at a time
    AddDot1x4(...);            // Compute 4 dot products
  }
}
```

- **Outer loop unrolling**: Jumps 4 columns per iteration
- **Inner loop**: Remains at stride 1, maintaining sequential memory access

Micro-kernel: AddDot1x4 The key innovation is the manual **4× loop unrolling** within the inner product computation:

```
for (p=0; p<k; p+=4) {
  // Load 4 elements from matrix A (same row, different columns)
  // Update 4 accumulation registers simultaneously
  c_00_reg += a_0p_reg * *bp0_pntr++;
  // ... repeated for all 4 columns
}
```

Critical optimizations: 1. **Register variables**: Using `register` keyword hints to keep accumulators in CPU registers 2. **Pointer arithmetic**: Separate pointers for each B column enable sequential access 3. **Manual unrolling**: Reduces loop control overhead from k iterations to k/4 iterations 4. **Reuse of A values**: Each `a_0p_reg` loads once but multiplies with 4 different B values

Memory Access Analysis

- **Matrix A**: Accessed with stride 1 (optimal spatial locality)
- **Matrix B**: Accessed with stride `ldb` (non-contiguous, but 4 columns processed in parallel)
- **Matrix C**: Updated once per 4×k operations (good computational intensity)

3. HPC Context

Performance Relevance

1. **Register Pressure Management**:
    - 5 register variables (`c_00_reg` to `c_03_reg` and `a_0p_reg`) fit comfortably in standard x86/ARM register files
    - Prevents spilling accumulators to slower cache/memory

2. **Instruction-Level Parallelism (ILP)**:

```
// Compiler can schedule these independently:
FMUL r1, rA, rB0  // Multiply A with B-column0
FMUL r2, rA, rB1  // Multiply same A with B-column1 (parallelizable)
FADD r3, r3, r1   // Accumulate to column0 result
FADD r4, r4, r2   // Accumulate to column1 result (parallelizable)
```

   - Multiple independent multiply-add operations enable **superscalar execution**
   - Modern CPUs can issue 2-4 FP operations per cycle with proper scheduling

3. **Loop Overhead Reduction**:
   - Original: k iterations with increment, compare, branch
   - Unrolled: k/4 iterations with same control logic
   - **4× reduction in branch misprediction penalties**

4. **Memory Bandwidth Efficiency**:
   - **Compute-to-memory ratio**: 8 FLOPs (4 multiplies + 4 adds) per 5 memory accesses
   - **Operational intensity**: ~1.6 FLOPs/byte (if double precision)
   - This approaches but doesn't saturate modern CPU capabilities (typically 4-8 FLOPs/byte peak)

Limitations and Next Steps

1. **Cache Locality Issues**:
   - No temporal reuse of B elements across different i iterations
   - Each B element loaded once, then discarded (poor cache utilization)
2. **Scalability Constraints**:
   - Only unrolls in column dimension (j)
   - Doesn't exploit register reuse in row dimension (i)
   - Limited to 4 parallel accumulations
3. **Future Optimization Paths**:
   - **Register blocking**: Unroll both i and j loops (e.g., 4×4 kernel)
   - **Cache blocking**: Add outer loops to operate on cache-sized blocks
   - **SIMD vectorization**: Replace scalar operations with AVX/NEON intrinsics
   - **Loop reordering**: Explore j-i-p or i-j-p variants for better locality

Educational Significance This code exemplifies the **transition from naive to optimized implementations** by demonstrating: - How manual loop transformations affect performance - The importance of minimizing loop overhead in tight computational kernels - The foundation upon which more sophisticated optimizations (like GEMM in BLAS libraries) are built - The trade-off between code complexity and performance gain

Key Insight: This optimization achieves 2-3× speedup over naive implementation but remains memory-bound. Further improvements require addressing cache and memory hierarchy constraints through blocking techniques.

Source Code Implementation

```
/* Create macros so that the matrices are stored in column-major order */

#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Routine for computing C = A * B + C */

void AddDot1x4( int, double *, int,  double *, int, double *, int )

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j;

  for ( j=0; j<n; j+=4 ){        /* Loop over the columns of C, unrolled by 4 */
    for ( i=0; i<m; i+=1 ){        /* Loop over the rows of C */
      /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
     one routine (four inner products) */

      AddDot1x4( k, &A( i,0 ), lda, &B( 0,j ), ldb, &C( i,j ), ldc );
    }
  }
}


void AddDot1x4( int k, double *a, int lda,  double *b, int ldb, double *c, int ldc )
{
  /* So, this routine computes four elements of C:

           C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ).

     Notice that this routine is called with c = C( i, j ) in the
     previous routine, so these are actually the elements

           C( i, j ), C( i, j+1 ), C( i, j+2 ), C( i, j+3 )

     in the original matrix C.
```

```
   We now unroll the loop */

int p;
register double
  /* hold contributions to
     C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ) */
     c_00_reg,   c_01_reg,   c_02_reg,   c_03_reg,
  /* holds A( 0, p ) */
     a_0p_reg;
double
  /* Point to the current elements in the four columns of B */
  *bp0_pntr, *bp1_pntr, *bp2_pntr, *bp3_pntr;

bp0_pntr = &B( 0, 0 );
bp1_pntr = &B( 0, 1 );
bp2_pntr = &B( 0, 2 );
bp3_pntr = &B( 0, 3 );

c_00_reg = 0.0;
c_01_reg = 0.0;
c_02_reg = 0.0;
c_03_reg = 0.0;

for ( p=0; p<k; p+=4 ){
  a_0p_reg = A( 0, p );

  c_00_reg += a_0p_reg * *bp0_pntr++;
  c_01_reg += a_0p_reg * *bp1_pntr++;
  c_02_reg += a_0p_reg * *bp2_pntr++;
  c_03_reg += a_0p_reg * *bp3_pntr++;

  a_0p_reg = A( 0, p+1 );

  c_00_reg += a_0p_reg * *bp0_pntr++;
  c_01_reg += a_0p_reg * *bp1_pntr++;
  c_02_reg += a_0p_reg * *bp2_pntr++;
  c_03_reg += a_0p_reg * *bp3_pntr++;

  a_0p_reg = A( 0, p+2 );

  c_00_reg += a_0p_reg * *bp0_pntr++;
  c_01_reg += a_0p_reg * *bp1_pntr++;
  c_02_reg += a_0p_reg * *bp2_pntr++;
  c_03_reg += a_0p_reg * *bp3_pntr++;

  a_0p_reg = A( 0, p+3 );
```

```
    c_00_reg += a_0p_reg * *bp0_pntr++;
    c_01_reg += a_0p_reg * *bp1_pntr++;
    c_02_reg += a_0p_reg * *bp2_pntr++;
    c_03_reg += a_0p_reg * *bp3_pntr++;
  }

  C( 0, 0 ) += c_00_reg;
  C( 0, 1 ) += c_01_reg;
  C( 0, 2 ) += c_02_reg;
  C( 0, 3 ) += c_03_reg;
}
```

File: MMult_1x4_9.c

Analysis of `MMult_1x4_9.c`: Optimized Matrix Multiplication Kernel

1. Role & Purpose

This file implements a **single-precision optimized kernel** for matrix-matrix multiplication (GEMM) within a larger HPC optimization study. It serves as an **intermediate optimization step** demonstrating:

- **Column-major storage** consistent with BLAS/LAPACK conventions
- **Loop unrolling** in both inner and outer loops
- **Register blocking** to maximize data reuse
- **Pointer-based indirect addressing** to reduce address calculation overhead

The kernel computes $\mathbf{C} = \mathbf{A} \times \mathbf{B} + \mathbf{C}$ (GEMM operation) where A is m×k, B is k×n, and C is m×n. This specific version targets the optimization of **single-row, multiple-column updates** (1×4 blocks of C).

2. Technical Details

Macro-Based Memory Access

```
#define A(i,j) a[ (j)*lda + (i) ]  // Column-major addressing
```

- **Column-major order**: Matrices are stored column-wise, consistent with Fortran/BLAS conventions
- **Leading dimension parameters** (`lda`, `ldb`, `ldc`): Allow submatrix operations by specifying storage stride
- **Macro abstraction**: Simplifies index calculations while maintaining readability

Nested Loop Structure

```
for (j=0; j<n; j+=4) {    // Process 4 columns at a time
  for (i=0; i<m; i+=1) {  // Process 1 row at a time
    AddDot1x4(...);        // Compute 1×4 block of C
  }
}
```

- **Outer loop unrolling**: Processes 4 columns of C per iteration (`j+=4`)
- **Inner loop**: Processes single rows (`i+=1`), creating 1×4 computational blocks
- **Blocking strategy**: The 1×4 blocking pattern is a compromise between register pressure and instruction-level parallelism

AddDot1x4 Kernel The core computational kernel employs several optimization techniques:

Register Allocation:

```
register double c_00_reg, c_01_reg, c_02_reg, c_03_reg, a_0p_reg;
```

- **Register variables**: Hint to compiler to keep intermediate values in CPU registers
- **Accumulator registers**: Four separate registers for C elements minimize memory traffic
- **A element register**: `a_0p_reg` holds current A element for reuse across 4 multiplications

Pointer-Based B Access:

```
double *bp0_pntr, *bp1_pntr, *bp2_pntr, *bp3_pntr;
bp0_pntr = &B(0,0);  // Points to current position in each B column
```

- **Column pointers**: Separate pointers for four columns of B enable sequential access
- **Pointer arithmetic**: Updates via `bp0_pntr+=4` avoid repeated index calculations
- **Indirect addressing**: `*bp0_pntr` dereferences pointers directly

Inner Loop Unrolling:

```
for (p=0; p<k; p+=4) {
  a_0p_reg = A(0,p);
  c_00_reg += a_0p_reg * *bp0_pntr;
  // ... 3 similar operations for other columns
  // Repeated for p+1, p+2, p+3 with offset pointer access
}
```

- **4-way unrolling**: Processes 4 elements of k-dimension per iteration
- **Reused A value**: Single A element multiplied with 4 consecutive B elements

- **Sequential memory access**: B pointers access contiguous memory (`*(bp0_pntr+1)`), enabling prefetching

3. HPC Context

Memory Hierarchy Optimization

- **Register tiling**: The 1×4 block size is chosen to fit accumulated C values in registers
- **Cache locality**: Access pattern for B is column-wise but with spatial locality within each column
- **A reuse**: Each A element is reused 4 times (for 4 B columns) while in register

Instruction-Level Parallelism (ILP)

- **Independent operations**: Four multiply-add chains can execute concurrently on superscalar processors
- **Reduced loop overhead**: 4× unrolling in k-loop decreases branch prediction penalties
- **Pointer arithmetic**: Moves address calculations out of innermost loop

Performance Limitations & Trade-offs

- **Register pressure**: 5 active registers may limit further unrolling on some architectures
- **Memory access pattern**: Still uses column-wise B access, which is suboptimal for row-major architectures
- **No SIMD utilization**: This scalar implementation doesn't exploit vector instructions (SSE/AVX)
- **Fixed blocking factor**: The 1×4 blocking may not be optimal for all cache hierarchies

Evolutionary Optimization Context This kernel represents a **middle ground** between naive implementation and fully optimized BLAS: - **Baseline**: Triple nested loops with $O(n^3)$ operations and poor cache utilization - **This version**: Unrolled loops + register blocking for ~2-4× speedup - **Next steps**: Typically followed by cache blocking, SIMD vectorization, and assembly tuning

Relevance to Modern HPC While not production-ready (lacks SIMD, dynamic blocking), this code demonstrates **fundamental optimization principles**: - **Minimize memory operations**: Via registers and pointer reuse - **Maximize operation density**: Via loop unrolling - **Sequential memory access**: For hardware prefetching benefits - **Abstraction through macros**: Maintains readability while enabling optimization

This implementation serves as an **excellent teaching example** of incremental optimization, showing measurable performance improvements while remaining pedagogically clear. In practice, production BLAS implementations would extend these concepts with architecture-specific optimizations, multithreading, and sophisticated prefetching strategies.

Source Code Implementation

```
/* Create macros so that the matrices are stored in column-major order */

#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Routine for computing C = A * B + C */

void AddDot1x4( int, double *, int,  double *, int, double *, int )

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j;

  for ( j=0; j<n; j+=4 ){        /* Loop over the columns of C, unrolled by 4 */
    for ( i=0; i<m; i+=1 ){        /* Loop over the rows of C */
      /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
     one routine (four inner products) */

      AddDot1x4( k, &A( i,0 ), lda, &B( 0,j ), ldb, &C( i,j ), ldc );
    }
  }
}


void AddDot1x4( int k, double *a, int lda,  double *b, int ldb, double *c, int ldc )
{
  /* So, this routine computes four elements of C:

           C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ).

     Notice that this routine is called with c = C( i, j ) in the
     previous routine, so these are actually the elements

           C( i, j ), C( i, j+1 ), C( i, j+2 ), C( i, j+3 )
```

```
   in the original matrix C.

   We next use indirect addressing */

int p;
register double
  /* hold contributions to
     C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ) */
     c_00_reg,   c_01_reg,   c_02_reg,   c_03_reg,
  /* holds A( 0, p ) */
     a_0p_reg;
double
  /* Point to the current elements in the four columns of B */
  *bp0_pntr, *bp1_pntr, *bp2_pntr, *bp3_pntr;

bp0_pntr = &B( 0, 0 );
bp1_pntr = &B( 0, 1 );
bp2_pntr = &B( 0, 2 );
bp3_pntr = &B( 0, 3 );

c_00_reg = 0.0;
c_01_reg = 0.0;
c_02_reg = 0.0;
c_03_reg = 0.0;

for ( p=0; p<k; p+=4 ){
  a_0p_reg = A( 0, p );

  c_00_reg += a_0p_reg * *bp0_pntr;
  c_01_reg += a_0p_reg * *bp1_pntr;
  c_02_reg += a_0p_reg * *bp2_pntr;
  c_03_reg += a_0p_reg * *bp3_pntr;

  a_0p_reg = A( 0, p+1 );

  c_00_reg += a_0p_reg * *(bp0_pntr+1);
  c_01_reg += a_0p_reg * *(bp1_pntr+1);
  c_02_reg += a_0p_reg * *(bp2_pntr+1);
  c_03_reg += a_0p_reg * *(bp3_pntr+1);

  a_0p_reg = A( 0, p+2 );

  c_00_reg += a_0p_reg * *(bp0_pntr+2);
  c_01_reg += a_0p_reg * *(bp1_pntr+2);
  c_02_reg += a_0p_reg * *(bp2_pntr+2);
  c_03_reg += a_0p_reg * *(bp3_pntr+2);
```

```
    a_0p_reg = A( 0, p+3 );

    c_00_reg += a_0p_reg * *(bp0_pntr+3);
    c_01_reg += a_0p_reg * *(bp1_pntr+3);
    c_02_reg += a_0p_reg * *(bp2_pntr+3);
    c_03_reg += a_0p_reg * *(bp3_pntr+3);

    bp0_pntr+=4;
    bp1_pntr+=4;
    bp2_pntr+=4;
    bp3_pntr+=4;
  }

  C( 0, 0 ) += c_00_reg;
  C( 0, 1 ) += c_01_reg;
  C( 0, 2 ) += c_02_reg;
  C( 0, 3 ) += c_03_reg;
}
```

Chapter 5: Vectorization (4x4 SIMD)

File: MMult_4x4_10.c

Analysis of MMult_4x4_10.c

1. Role & Purpose

This file implements a **highly optimized matrix multiplication kernel** for computing $\mathbf{C} = \mathbf{A} \times \mathbf{B} + \mathbf{C}$ using **SIMD (Single Instruction, Multiple Data) intrinsics**. It serves as a **performance demonstration** within an optimization tutorial series, showcasing how to leverage **SSE (Streaming SIMD Extensions)** for double-precision floating-point operations. The code specifically targets **4×4 block matrix multiplication** through **loop unrolling** and **vector register utilization**, forming a critical building block for high-performance linear algebra libraries.

2. Technical Details

Matrix Storage and Access Macros

```
#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]
```

- **Column-major ordering**: Elements are stored column-wise, consistent with Fortran and BLAS conventions.
- **Leading dimensions (lda, ldb, ldc)**: Allow handling of submatrices within larger matrices.
- **Macro-based indexing**: Enables cleaner code while maintaining pointer arithmetic efficiency.

Main Multiplication Routine

```
void MY_MMult( int m, int n, int k, double *a, int lda,
               double *b, int ldb, double *c, int ldc )
{
  for ( j=0; j<n; j+=4 ){        /* Loop over columns of C, unrolled by 4 */
    for ( i=0; i<m; i+=4 ){        /* Loop over rows of C */
      AddDot4x4( k, &A( i,0 ), lda, &B( 0,j ), ldb, &C( i,j ), ldc );
    }
  }
}
```

- **Blocked computation**: Processes C in 4×4 blocks.
- **Pointer arithmetic**: Passes pointers to submatrices to the inner kernel.
- **Unrolled loops**: Reduces loop overhead and enables better instruction scheduling.

SIMD Implementation with SSE

```
typedef union {
  __m128d v;
  double d[2];
} v2df_t;
```

- **Vector type definition**: Union allows access to SSE register as either two doubles or a 128-bit vector.
- **__m128d**: SSE data type holding two double-precision values.

Core Computational Kernel

```
void AddDot4x4( int k, double *a, int lda, double *b, int ldb, double *c, int ldc )
```

- **Register strategy**: Uses 12 vector registers to hold intermediate results:
  - 8 accumulators for C matrix elements (paired as two rows per register)
  - 2 registers for loading columns of A
  - 4 registers for broadcasting elements of B
- **Key SSE intrinsics**:
  - `_mm_setzero_pd()`: Creates vector of two zeros.
  - `_mm_load_pd()`: Loads 128-bit (two doubles) from aligned memory.
  - `_mm_loaddup_pd()`: Broadcasts a single double to both vector lanes.

  - Vector operations: Uses C extensions for `+` and `*` on `__m128d` types.
- **Computation pattern**:
  1. Load two rows of A (four elements total) into two registers
  2. Broadcast single elements of B into four registers
  3. Perform vector multiply-add operations
  4. Accumulate results in eight register pairs
  5. Store final results back to C matrix
- **Memory access patterns**:
  - **A matrix**: Contiguous access within columns (column-major advantage)
  - **B matrix**: Sequential access down columns with element broadcasting
  - **C matrix**: Register accumulation minimizes memory writes

3. HPC Context

Performance Optimization Techniques

1. **SIMD Parallelism**: Leverages 128-bit SSE registers to perform **two double-precision operations per instruction**, doubling theoretical FLOP/s compared to scalar code.
2. **Register Blocking**: The 4×4 block size is chosen to:
   - **Minimize memory traffic** by reusing loaded values
   - **Fit comfortably in SSE registers** (8 accumulators + working registers)
   - **Match typical cache line sizes** (64 bytes for two columns of A)
3. **Loop Unrolling Benefits**:
   - **Reduces loop overhead** from 16 inner loops (4×4) to one kernel call
   - **Enables better instruction scheduling** by exposing more operations to compiler
   - **Facilitates register reuse** across multiple computations
4. **Memory Hierarchy Optimization**:
   - **Spatial locality**: Contiguous memory accesses for A matrix columns
   - **Temporal locality**: B elements broadcasted and reused across A rows
   - **Register tiling**: 4×4 block keeps working set in fastest memory (registers)

Limitations and Trade-offs

- **Fixed block size**: Inflexible for non-multiple-of-4 dimensions (requires edge handling)

- **SSE2 limitation**: Only 2-wide double precision, whereas AVX/AVX-512 offer 4/8-wide
- **No prefetching**: Could benefit from explicit cache line prefetch instructions
- **Alignment requirements**: `_mm_load_pd()` requires 16-byte alignment
- **No FMA (Fused Multiply-Add)**: Uses separate multiply and add operations

Educational Value This code represents a **critical milestone** in matrix multiplication optimization: 1. **Demonstrates SIMD principles**: Shows how to vectorize inner products 2. **Illustrates data packing strategies**: Pairs rows for vector processing 3. **Highlights register pressure management**: Careful allocation of limited vector registers 4. **Shows the transition** from naive triple loops to architecture-aware implementations

This implementation typically achieves **substantial speedup** over naive C code (often 4-8×), serving as foundation for more advanced optimizations like cache blocking, AVX extensions, and algorithmic improvements for very large matrices.

Source Code Implementation

```
/* Create macros so that the matrices are stored in column-major order */

#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Routine for computing C = A * B + C */

void AddDot4x4( int, double *, int, double *, int, double *, int );

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j;

  for ( j=0; j<n; j+=4 ){        /* Loop over the columns of C, unrolled by 4 */
    for ( i=0; i<m; i+=4 ){        /* Loop over the rows of C */
      /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
     one routine (four inner products) */

      AddDot4x4( k, &A( i,0 ), lda, &B( 0,j ), ldb, &C( i,j ), ldc );
    }
```

```
  }
}

#include <mmintrin.h>
#include <xmmintrin.h>  // SSE
#include <pmmintrin.h>  // SSE2
#include <emmintrin.h>  // SSE3

typedef union
{
  __m128d v;
  double d[2];
} v2df_t;

void AddDot4x4( int k, double *a, int lda,  double *b, int ldb, double *c, int ldc )
{
  /* So, this routine computes a 4x4 block of matrix A

           C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ).
           C( 1, 0 ), C( 1, 1 ), C( 1, 2 ), C( 1, 3 ).
           C( 2, 0 ), C( 2, 1 ), C( 2, 2 ), C( 2, 3 ).
           C( 3, 0 ), C( 3, 1 ), C( 3, 2 ), C( 3, 3 ).

     Notice that this routine is called with c = C( i, j ) in the
     previous routine, so these are actually the elements

           C( i  , j ), C( i  , j+1 ), C( i  , j+2 ), C( i  , j+3 )
           C( i+1, j ), C( i+1, j+1 ), C( i+1, j+2 ), C( i+1, j+3 )
           C( i+2, j ), C( i+2, j+1 ), C( i+2, j+2 ), C( i+2, j+3 )
           C( i+3, j ), C( i+3, j+1 ), C( i+3, j+2 ), C( i+3, j+3 )

     in the original matrix C

     And now we use vector registers and instructions */

  int p;

  v2df_t
    c_00_c_10_vreg,    c_01_c_11_vreg,    c_02_c_12_vreg,    c_03_c_13_vreg,
    c_20_c_30_vreg,    c_21_c_31_vreg,    c_22_c_32_vreg,    c_23_c_33_vreg,
    a_0p_a_1p_vreg,
    a_2p_a_3p_vreg,
    b_p0_vreg, b_p1_vreg, b_p2_vreg, b_p3_vreg;

  double
    /* Point to the current elements in the four columns of B */
```

```
  *b_p0_pntr, *b_p1_pntr, *b_p2_pntr, *b_p3_pntr;

b_p0_pntr = &B( 0, 0 );
b_p1_pntr = &B( 0, 1 );
b_p2_pntr = &B( 0, 2 );
b_p3_pntr = &B( 0, 3 );

c_00_c_10_vreg.v = _mm_setzero_pd();
c_01_c_11_vreg.v = _mm_setzero_pd();
c_02_c_12_vreg.v = _mm_setzero_pd();
c_03_c_13_vreg.v = _mm_setzero_pd();
c_20_c_30_vreg.v = _mm_setzero_pd();
c_21_c_31_vreg.v = _mm_setzero_pd();
c_22_c_32_vreg.v = _mm_setzero_pd();
c_23_c_33_vreg.v = _mm_setzero_pd();

for ( p=0; p<k; p++ ){
  a_0p_a_1p_vreg.v = _mm_load_pd( (double *) &A( 0, p ) );
  a_2p_a_3p_vreg.v = _mm_load_pd( (double *) &A( 2, p ) );

  b_p0_vreg.v = _mm_loaddup_pd( (double *) b_p0_pntr++ );   /* load and duplicate */
  b_p1_vreg.v = _mm_loaddup_pd( (double *) b_p1_pntr++ );   /* load and duplicate */
  b_p2_vreg.v = _mm_loaddup_pd( (double *) b_p2_pntr++ );   /* load and duplicate */
  b_p3_vreg.v = _mm_loaddup_pd( (double *) b_p3_pntr++ );   /* load and duplicate */

  /* First row and second rows */
  c_00_c_10_vreg.v += a_0p_a_1p_vreg.v * b_p0_vreg.v;
  c_01_c_11_vreg.v += a_0p_a_1p_vreg.v * b_p1_vreg.v;
  c_02_c_12_vreg.v += a_0p_a_1p_vreg.v * b_p2_vreg.v;
  c_03_c_13_vreg.v += a_0p_a_1p_vreg.v * b_p3_vreg.v;

  /* Third and fourth rows */
  c_20_c_30_vreg.v += a_2p_a_3p_vreg.v * b_p0_vreg.v;
  c_21_c_31_vreg.v += a_2p_a_3p_vreg.v * b_p1_vreg.v;
  c_22_c_32_vreg.v += a_2p_a_3p_vreg.v * b_p2_vreg.v;
  c_23_c_33_vreg.v += a_2p_a_3p_vreg.v * b_p3_vreg.v;
}

C( 0, 0 ) += c_00_c_10_vreg.d[0];  C( 0, 1 ) += c_01_c_11_vreg.d[0];
C( 0, 2 ) += c_02_c_12_vreg.d[0];  C( 0, 3 ) += c_03_c_13_vreg.d[0];

C( 1, 0 ) += c_00_c_10_vreg.d[1];  C( 1, 1 ) += c_01_c_11_vreg.d[1];
C( 1, 2 ) += c_02_c_12_vreg.d[1];  C( 1, 3 ) += c_03_c_13_vreg.d[1];

C( 2, 0 ) += c_20_c_30_vreg.d[0];  C( 2, 1 ) += c_21_c_31_vreg.d[0];
C( 2, 2 ) += c_22_c_32_vreg.d[0];  C( 2, 3 ) += c_23_c_33_vreg.d[0];
```

```
  C( 3, 0 ) += c_20_c_30_vreg.d[1];  C( 3, 1 ) += c_21_c_31_vreg.d[1];
  C( 3, 2 ) += c_22_c_32_vreg.d[1];  C( 3, 3 ) += c_23_c_33_vreg.d[1];
}
```

File: MMult_4x4_3.c

Analysis of MMult_4x4_3.c

1. Role & Purpose

This file implements **matrix multiplication optimization** using a **blocking and unrolling strategy** for computing C = A × B + C. It serves as an intermediate step in the **performance optimization hierarchy**, demonstrating how to restructure computation to improve **data locality** and reduce **loop overhead**. The code specifically:

- Implements **4×4 blocking** of the output matrix C
- Uses **loop unrolling** in both row and column dimensions
- Maintains **column-major storage** consistent with BLAS/LAPACK conventions
- Demonstrates **manual optimization techniques** that precede SIMD vectorization

2. Technical Details

Matrix Storage and Indexing

```
#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]
```

- These macros implement **column-major ordering**: consecutive elements in memory belong to the same column
- `lda`, `ldb`, `ldc` are **leading dimensions** (typically equal to number of rows for non-submatrices)
- The indexing `(j)*lda + (i)` calculates: `base_address + column_index*rows_per_column + row_index`

Main Multiplication Routine

```
void MY_MMult( int m, int n, int k, double *a, int lda,
               double *b, int ldb, double *c, int ldc )
{
  for ( j=0; j<n; j+=4 ){        /* Loop over columns of C, unrolled by 4 */
    for ( i=0; i<m; i+=4 ){      /* Loop over rows of C */
```

```
      AddDot4x4( k, &A( i,0 ), lda, &B( 0,j ), ldb, &C( i,j ), ldc );
    }
  }
}
```

- **Loop structure**: Outer loops traverse C in 4×4 blocks
- **Unrolling factor**: 4 in both dimensions reduces loop overhead by factor of 16
- **Block computation**: Each iteration computes a **4×4 submatrix** of C

Block Computation Implementation

```
void AddDot4x4( int k, double *a, int lda, double *b, int ldb,
                double *c, int ldc )
{
  /* Computes 4×4 block: C[i:i+3, j:j+3] += A[i:i+3, :] × B[:, j:j+3] */

  /* First row of block */
  AddDot( k, &A( 0, 0 ), lda, &B( 0, 0 ), &C( 0, 0 ) );
  AddDot( k, &A( 0, 0 ), lda, &B( 0, 1 ), &C( 0, 1 ) );
  /* ... 14 more similar calls */
}
```

- **Explicit unrolling**: 16 separate calls to `AddDot` compute each element
- **Data reuse**: Each row of A (`&A(r, 0)`) reused for 4 columns of B
- **Memory access**: Accesses are **strided** for A (stride = lda), **contiguous** for B columns

Inner Product Kernel

```
void AddDot( int k, double *x, int incx, double *y, double *gamma )
{
  /* gamma := x' * y + gamma */
  for ( p=0; p<k; p++ ){
    *gamma += X( p ) * y[ p ];
  }
}
```

- **Dot product**: Computes $\sum$(x[p] × y[p]) for p = 0 to k-1
- **Stride handling**: `X(p)` macro expands to `x[p*incx]` for strided access
- **Accumulation**: Directly accumulates into output location

Critical Memory Access Patterns

```
For computing C[i,j]:
  A accesses: A[i,0], A[i,1], A[i,2], ... with stride = lda
  B accesses: B[0,j], B[1,j], B[2,j], ... contiguous (stride = 1)
```

- **Spatial locality**: B accesses benefit from **cache line utilization**

- **Temporal locality**: Each A row reused across 4 B columns within block

3. HPC Context

Performance Characteristics

1. **Loop Overhead Reduction**
   - Unrolling reduces branch instructions by factor of ~16
   - Enables better **instruction scheduling** and **pipelining**
   - Compiler has more visibility for **register allocation**
2. **Data Locality Improvements**
   - **Blocking** keeps working set small enough for registers/L1 cache
   - 4×4 blocking = 16 output elements computed together
   - Each A row (4 elements at a time) stays in register/load unit
3. **Memory Hierarchy Optimization**
   - **Register reuse**: Each A element loaded once, used 4 times
   - **Cache line utilization**: Complete cache lines of B consumed
   - **Prefetching**: Regular access patterns enable hardware prefetching
4. **Limitations and Next Steps**
   - **No SIMD vectorization**: Still uses scalar operations
   - **Register pressure**: 16 accumulators may exceed register file
   - **Fixed block size**: Not adaptive to different cache sizes
   - **No packing**: Matrices not reorganized for contiguous access

Performance Trade-offs

Advantage	Cost/Limitation
Reduced loop overhead	Increased code size
Better instruction scheduling	Fixed, non-adaptive blocking
Temporal locality within block	Still suboptimal cache reuse
Explicit control flow	Not portable across architectures

Evolutionary Context

This represents **Optimization Level 3** in a typical optimization sequence: 1. Naive triple loop → 2. Loop order optimization → **3. Blocking/Unrolling** → 4. SIMD vectorization → 5. Cache-aware blocking → 6. Multi-threading

Key Insights for Students

- **Blocking factor** (4) balances register usage and reduction of loop overhead
- **Explicit unrolling** makes data dependencies visible to compiler
- **Memory access patterns** more important than arithmetic intensity at this stage

- This approach prepares for **SIMD vectorization** (next optimization step)

Performance Expectations

- **Speedup over naive**: Typically 2-4× on modern CPUs
- **Bottleneck**: Memory bandwidth for large matrices
- **Optimal use case**: Moderate-sized matrices that fit in L2/L3 cache
- **Foundation**: Essential building block for more advanced optimizations

This implementation demonstrates fundamental HPC principles: **locality enhancement**, **loop restructuring**, and **explicit control** over computation patterns. While not peak-performance code, it establishes patterns essential for understanding advanced optimization techniques.

Source Code Implementation

```
/* Create macros so that the matrices are stored in column-major order */

#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Routine for computing C = A * B + C */

void AddDot( int, double *, int, double *, double * );

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j;

  for ( j=0; j<n; j+=4 ){        /* Loop over the columns of C, unrolled by 4 */
    for ( i=0; i<m; i+=4 ){        /* Loop over the rows of C */
      /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
     one routine (four inner products) */

      AddDot4x4( k, &A( i,0 ), lda, &B( 0,j ), ldb, &C( i,j ), ldc );
    }
  }
}


void AddDot4x4( int k, double *a, int lda,  double *b, int ldb, double *c, int ldc )
{
```

```
  /* So, this routine computes a 4x4 block of matrix A

           C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ).
           C( 1, 0 ), C( 1, 1 ), C( 1, 2 ), C( 1, 3 ).
           C( 2, 0 ), C( 2, 1 ), C( 2, 2 ), C( 2, 3 ).
           C( 3, 0 ), C( 3, 1 ), C( 3, 2 ), C( 3, 3 ).

     Notice that this routine is called with c = C( i, j ) in the
     previous routine, so these are actually the elements

           C( i  , j ), C( i  , j+1 ), C( i  , j+2 ), C( i  , j+3 )
           C( i+1, j ), C( i+1, j+1 ), C( i+1, j+2 ), C( i+1, j+3 )
           C( i+2, j ), C( i+2, j+1 ), C( i+2, j+2 ), C( i+2, j+3 )
           C( i+3, j ), C( i+3, j+1 ), C( i+3, j+2 ), C( i+3, j+3 )

     in the original matrix C */

  /* First row */
  AddDot( k, &A( 0, 0 ), lda, &B( 0, 0 ), &C( 0, 0 ) );
  AddDot( k, &A( 0, 0 ), lda, &B( 0, 1 ), &C( 0, 1 ) );
  AddDot( k, &A( 0, 0 ), lda, &B( 0, 2 ), &C( 0, 2 ) );
  AddDot( k, &A( 0, 0 ), lda, &B( 0, 3 ), &C( 0, 3 ) );

  /* Second row */
  AddDot( k, &A( 1, 0 ), lda, &B( 0, 0 ), &C( 1, 0 ) );
  AddDot( k, &A( 1, 0 ), lda, &B( 0, 1 ), &C( 1, 1 ) );
  AddDot( k, &A( 1, 0 ), lda, &B( 0, 2 ), &C( 1, 2 ) );
  AddDot( k, &A( 1, 0 ), lda, &B( 0, 3 ), &C( 1, 3 ) );

  /* Third row */
  AddDot( k, &A( 2, 0 ), lda, &B( 0, 0 ), &C( 2, 0 ) );
  AddDot( k, &A( 2, 0 ), lda, &B( 0, 1 ), &C( 2, 1 ) );
  AddDot( k, &A( 2, 0 ), lda, &B( 0, 2 ), &C( 2, 2 ) );
  AddDot( k, &A( 2, 0 ), lda, &B( 0, 3 ), &C( 2, 3 ) );

  /* Four row */
  AddDot( k, &A( 3, 0 ), lda, &B( 0, 0 ), &C( 3, 0 ) );
  AddDot( k, &A( 3, 0 ), lda, &B( 0, 1 ), &C( 3, 1 ) );
  AddDot( k, &A( 3, 0 ), lda, &B( 0, 2 ), &C( 3, 2 ) );
  AddDot( k, &A( 3, 0 ), lda, &B( 0, 3 ), &C( 3, 3 ) );
}


/* Create macro to let X( i ) equal the ith element of x */

#define X(i) x[ (i)*incx ]
```

```c
void AddDot( int k, double *x, int incx,  double *y, double *gamma )
{
  /* compute gamma := x' * y + gamma with vectors x and y of length n.

     Here x starts at location x with increment (stride) incx and y starts at location y and
  */

  int p;

  for ( p=0; p<k; p++ ){
    *gamma += X( p ) * y[ p ];
  }
}
```

File: MMult_4x4_4.c

Analysis of MMult_4x4_4.c

1. Role & Purpose

This file implements **matrix multiplication** within a systematic optimization framework. Specifically, it demonstrates **loop unrolling** at the 4×4 block level while maintaining **column-major storage** compatibility (consistent with BLAS/LAPACK conventions). The primary purpose is to serve as an **intermediate optimization step** in a pedagogical progression from naive triple-loop implementations toward more advanced techniques like cache blocking and vectorization. By processing 4×4 blocks of matrix C, it reduces loop overhead and enables better **instruction-level parallelism (ILP)** while establishing a foundation for subsequent memory hierarchy optimizations.

2. Technical Details

Storage Order and Access Macros

```c
#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]
```

- **Column-major order** is enforced: element `(i,j)` is stored at offset `j*ld + i` where `ld` is the leading dimension (typically the row count).
- These macros abstract indexing, making the code dimension-agnostic while maintaining correct stride calculations for submatrices.

Main Multiplication Routine

```
void MY_MMult( int m, int n, int k, ... )
{
  for ( j=0; j<n; j+=4 ) {
    for ( i=0; i<m; i+=4 ) {
      AddDot4x4( k, &A( i,0 ), lda, &B( 0,j ), ldb, &C( i,j ), ldc );
    }
  }
}
```

- **Loop structure**: The outer loops traverse matrix C in 4×4 blocks (`j` over columns, `i` over rows).
- **Block granularity**: Each iteration computes a full 4×4 submatrix of C using all corresponding rows of A and columns of B.
- **Memory access pattern**: For a fixed block `C(i:i+3, j:j+3)`, the routine accesses:
  - A contiguous slice of four rows from A: `A(i:i+3, 0:k-1)`
  - Four contiguous columns from B: `B(0:k-1, j:j+3)`
- **Parameter passing**: Pointers are passed to the first element of each block, with leading dimensions to maintain correct striding.

Block Computation Kernel

```
void AddDot4x4( int k, ... )
{
  // 16 separate dot product loops
  for ( p=0; p<k; p++ ) {
    C( 0, 0 ) += A( 0, p ) * B( p, 0 );
  }
  // ... repeated for all 4×4 elements
}
```

- **Fully unrolled inner loops**: Each of the 16 output elements has its own explicit dot product loop over `p`.
- **Inline computation**: The commented `AddDot` calls show the manual inlining performed—replacing function calls with explicit loops.
- **Index arithmetic**: Inside `AddDot4x4`, indices are relative to the block start (0–3 rather than i–i+3).

Key Implementation Points

- **No temporal locality optimization**: Each element of A and B is loaded multiple times across different dot products.
- **Fixed unrolling factor**: Assumes dimensions are multiples of 4 (no edge-case handling shown).
- **Direct memory access**: All loads/stores go through the macro indexing without explicit register promotion.

3. HPC Context

Performance Strengths

1. **Reduced loop overhead**: The outer loops run `(m/4)*(n/4)` iterations instead of `m*n`, cutting branch prediction misses and loop counter updates.
2. **Improved instruction scheduling**: The compiler can better pipeline the 16 independent accumulation loops when they are explicitly unrolled.
3. **Foundation for vectorization**: The 4×4 structure naturally aligns with **SIMD register widths** (e.g., 256-bit AVX registers hold four doubles).
4. **Memory access patterns**: Within each 4×4 block, accesses to B are contiguous (`B(p,0:3)`), enabling potential **cache line utilization**.

Limitations and Optimization Pathways

1. **No cache blocking**: The entire `k` dimension is traversed for each block, causing repeated memory accesses. Optimal implementations would add a `K`-block loop.
2. **Register underutilization**: The 16 accumulators could be held in registers, but this implementation writes to memory each iteration. **Register tiling** would minimize stores.
3. **No vectorization**: The dot product loops are scalar. Modern implementations would use **AVX/AVX2 intrinsics** to compute four multiplies per instruction.
4. **No prefetching**: Hardware prefetching works better with predictable strides, but explicit software prefetching could help.
5. **Fixed block size**: A tunable block size parameter would allow architecture-specific optimization (e.g., 6×8 for AVX-512).

Educational Significance This code sits at a critical transition point in optimization: - It demonstrates **manual loop unrolling**, a fundamental technique. - It shows the **computational intensity** pattern: 16 flops per `p` iteration with 20 memory accesses (16 loads + 4 stores), giving an operational intensity of 0.8 flops/byte. - It sets up for **cache-aware blocking** (next logical step) where the `k`-loop would be tiled to fit L1/L2 caches. - The 4×4 blocking is a classic precursor to **SIMD vectorization**, where each row's four dot products could become a single vector operation.

This implementation represents the **middle ground** between naive algorithms and highly optimized BLAS kernels—ideal for teaching the incremental optimization process in HPC.

Source Code Implementation

```
/* Create macros so that the matrices are stored in column-major order */

#define A(i,j) a[ (j)*lda + (i) ]
```

```
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Routine for computing C = A * B + C */

void AddDot4x4( int, double *, int, double *, int, double *, int );
void AddDot( int, double *, int, double *, double * );

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j;

  for ( j=0; j<n; j+=4 ){        /* Loop over the columns of C, unrolled by 4 */
    for ( i=0; i<m; i+=4 ){        /* Loop over the rows of C */
      /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
     one routine (four inner products) */

      AddDot4x4( k, &A( i,0 ), lda, &B( 0,j ), ldb, &C( i,j ), ldc );
    }
  }
}


void AddDot4x4( int k, double *a, int lda,  double *b, int ldb, double *c, int ldc )
{
  /* So, this routine computes a 4x4 block of matrix A

           C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ).
           C( 1, 0 ), C( 1, 1 ), C( 1, 2 ), C( 1, 3 ).
           C( 2, 0 ), C( 2, 1 ), C( 2, 2 ), C( 2, 3 ).
           C( 3, 0 ), C( 3, 1 ), C( 3, 2 ), C( 3, 3 ).

     Notice that this routine is called with c = C( i, j ) in the
     previous routine, so these are actually the elements

           C( i  , j ), C( i  , j+1 ), C( i  , j+2 ), C( i  , j+3 )
           C( i+1, j ), C( i+1, j+1 ), C( i+1, j+2 ), C( i+1, j+3 )
           C( i+2, j ), C( i+2, j+1 ), C( i+2, j+2 ), C( i+2, j+3 )
           C( i+3, j ), C( i+3, j+1 ), C( i+3, j+2 ), C( i+3, j+3 )

     in the original matrix C

     In this version, we "inline" AddDot */
```

```
int p;

/* First row */
//  AddDot( k, &A( 0, 0 ), lda, &B( 0, 0 ), &C( 0, 0 ) );
for ( p=0; p<k; p++ ){
  C( 0, 0 ) += A( 0, p ) * B( p, 0 );
}
//  AddDot( k, &A( 0, 0 ), lda, &B( 0, 1 ), &C( 0, 1 ) );
for ( p=0; p<k; p++ ){
  C( 0, 1 ) += A( 0, p ) * B( p, 1 );
}
//  AddDot( k, &A( 0, 0 ), lda, &B( 0, 2 ), &C( 0, 2 ) );
for ( p=0; p<k; p++ ){
  C( 0, 2 ) += A( 0, p ) * B( p, 2 );
}
//  AddDot( k, &A( 0, 0 ), lda, &B( 0, 3 ), &C( 0, 3 ) );
for ( p=0; p<k; p++ ){
  C( 0, 3 ) += A( 0, p ) * B( p, 3 );
}

/* Second row */
//  AddDot( k, &A( 1, 0 ), lda, &B( 0, 0 ), &C( 1, 0 ) );
for ( p=0; p<k; p++ ){
  C( 1, 0 ) += A( 1, p ) * B( p, 0 );
}
//  AddDot( k, &A( 1, 0 ), lda, &B( 0, 1 ), &C( 1, 1 ) );
for ( p=0; p<k; p++ ){
  C( 1, 1 ) += A( 1, p ) * B( p, 1 );
}
//  AddDot( k, &A( 1, 0 ), lda, &B( 0, 2 ), &C( 1, 2 ) );
for ( p=0; p<k; p++ ){
  C( 1, 2 ) += A( 1, p ) * B( p, 2 );
}
//  AddDot( k, &A( 1, 0 ), lda, &B( 0, 3 ), &C( 1, 3 ) );
for ( p=0; p<k; p++ ){
  C( 1, 3 ) += A( 1, p ) * B( p, 3 );
}

/* Third row */
//  AddDot( k, &A( 2, 0 ), lda, &B( 0, 0 ), &C( 2, 0 ) );
for ( p=0; p<k; p++ ){
  C( 2, 0 ) += A( 2, p ) * B( p, 0 );
}
//  AddDot( k, &A( 2, 0 ), lda, &B( 0, 1 ), &C( 2, 1 ) );
for ( p=0; p<k; p++ ){
  C( 2, 1 ) += A( 2, p ) * B( p, 1 );
```

```
  }
  //  AddDot( k, &A( 2, 0 ), lda, &B( 0, 2 ), &C( 2, 2 ) );
  for ( p=0; p<k; p++ ){
    C( 2, 2 ) += A( 2, p ) * B( p, 2 );
  }
  //  AddDot( k, &A( 2, 0 ), lda, &B( 0, 3 ), &C( 2, 3 ) );
  for ( p=0; p<k; p++ ){
    C( 2, 3 ) += A( 2, p ) * B( p, 3 );
  }

  /* Four row */
  //  AddDot( k, &A( 3, 0 ), lda, &B( 0, 0 ), &C( 3, 0 ) );
  for ( p=0; p<k; p++ ){
    C( 3, 0 ) += A( 3, p ) * B( p, 0 );
  }
  //  AddDot( k, &A( 3, 0 ), lda, &B( 0, 1 ), &C( 3, 1 ) );
  for ( p=0; p<k; p++ ){
    C( 3, 1 ) += A( 3, p ) * B( p, 1 );
  }
  //  AddDot( k, &A( 3, 0 ), lda, &B( 0, 2 ), &C( 3, 2 ) );
  for ( p=0; p<k; p++ ){
    C( 3, 2 ) += A( 3, p ) * B( p, 2 );
  }
  //  AddDot( k, &A( 3, 0 ), lda, &B( 0, 3 ), &C( 3, 3 ) );
  for ( p=0; p<k; p++ ){
    C( 3, 3 ) += A( 3, p ) * B( p, 3 );
  }
}
```

File: MMult_4x4_5.c

Analysis of `MMult_4x4_5.c`: High-Performance Matrix Multiplication Kernel

1. Role & Purpose

This file implements **a blocked matrix multiplication kernel** within a performance optimization study. Its primary purpose is to demonstrate **register-level tiling** by computing **4×4 output blocks** of matrix C through **loop unrolling**. This represents an intermediate step toward achieving **near-peak floating-point performance** by:

- **Increasing arithmetic intensity** (FLOPs per memory access)
- **Exploiting data locality** through register reuse

- **Reducing loop overhead** via manual unrolling
- **Providing a foundation** for subsequent SIMD vectorization

The code operates within a **larger optimization framework** where different blocking strategies (1×4, 4×4, 8×8, etc.) are systematically compared to understand their performance characteristics on modern CPU architectures.

2. Technical Details

Macro Definitions and Memory Layout

```
#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]
```

- **Column-major ordering**: Matrices are stored column-wise, consistent with FORTRAN and LAPACK conventions
- **Leading dimension parameters** (`lda`, `ldb`, `ldc`): Support submatrices and non-contiguous memory access patterns
- **Macro abstraction**: Simplifies index calculation while maintaining clarity

Main Kernel Structure

```
void MY_MMult( int m, int n, int k, double *a, int lda,
               double *b, int ldb, double *c, int ldc )
{
  for ( j=0; j<n; j+=4 ){        /* Columns of C, unrolled by 4 */
    for ( i=0; i<m; i+=4 ){      /* Rows of C, unrolled by 4 */
      AddDot4x4( k, &A(i,0), lda, &B(0,j), ldb, &C(i,j), ldc );
    }
  }
}
```

- **Outer loop blocking**: Processes C in **4×4 tiles** (j-loop over columns, i-loop over rows)
- **Kernel invocation**: Each tile computed by `AddDot4x4` with appropriate pointer arithmetic

Core Computation: `AddDot4x4`

```
void AddDot4x4( int k, double *a, int lda, double *b, int ldb,
                double *c, int ldc )
{
  for ( p=0; p<k; p++ ){
    /* First row of 4x4 block */
    C(0,0) += A(0,p) * B(p,0);  C(0,1) += A(0,p) * B(p,1);
    C(0,2) += A(0,p) * B(p,2);  C(0,3) += A(0,p) * B(p,3);
```

```
    /* Similar expansions for rows 1-3 */
    C(1,0) += A(1,p) * B(p,0);  /* ... */ C(1,3) += A(1,p) * B(p,3);
    C(2,0) += A(2,p) * B(p,0);  /* ... */ C(2,3) += A(2,p) * B(p,3);
    C(3,0) += A(3,p) * B(p,0);  /* ... */ C(3,3) += A(3,p) * B(p,3);
  }
}
```

- **Manual unrolling**: The inner `k`-loop remains, but the 16 output accumulations are explicitly coded
- **Fused operations**: Each iteration loads **one column of A** (4 elements) and **one row of B** (4 elements), computing 16 multiply-add operations
- **Register reuse**: Each loaded A element (`A(r,p)`) is used 4 times; each B element (`B(p,c)`) is used 4 times
- **No intermediate storage**: Accumulation happens directly into C elements in memory

Algorithmic Complexity & Access Patterns

- **Total operations**: 2×4×4×k = 32k FLOPs per 4×4 block (k multiply-add pairs)
- **Memory accesses**:
  - A: 4k elements loaded
  - B: 4k elements loaded
  - C: 16 elements loaded and stored (assuming RMW)
- **Arithmetic intensity**: ~32k/(8k+32) 4 FLOPs/byte (theoretical peak)

3. HPC Context

Performance Characteristics

1. **Register Pressure Management**:
   - **Explicit unrolling** allows the compiler to allocate 16 accumulator variables to registers
   - **No temporary arrays** minimizes register spilling to cache
   - However, **32 register variables** (16 for C, 4 for A row, 4 for B column) may approach architectural limits
2. **Data Locality Optimization**:
   - **Register tiling**: 4×4 blocking fits typical register files (e.g., 16-32 FP registers)
   - **Temporal reuse**: Each A element reused 4×, each B element reused 4× within the inner loop
   - **Spatial locality**: Contiguous access in A (column) and B (row) enables efficient cache line utilization
3. **Instruction-Level Parallelism (ILP)**:

- **Dependency breaking**: 16 independent multiply-add chains enable out-of-order execution
- **Loop-carried dependencies** only through C accumulations, but these are independent across elements
- **Compiler optimization potential**: Unrolled operations can be scheduled for maximum pipeline utilization

4. **Memory Hierarchy Implications**:
   - **L1 cache friendly**: Working set fits easily (4k doubles = 32KB when k=1024)
   - **Prefetching effectiveness**: Predictable stride-1 access in A and B
   - **Write-combining**: 16 C accumulations may benefit from write buffering

Limitations and Evolution Path

1. **Missing SIMD Vectorization**:
   - The code computes scalar operations; modern versions would use **AVX/AVX2 intrinsics** to process 4-8 elements simultaneously
   - Next optimization step: Replace `C(r,0..3)` updates with `_mm256_fmadd_pd()` calls
2. **Cache Blocking Absent**:
   - This is **register-level blocking only**
   - For larger matrices, **L1/L2 cache blocking** (tiling) is necessary to avoid thrashing
   - Optimal hierarchy: Register tiles → L1 cache tiles → L2 cache tiles
3. **Performance Ceiling**:
   - **Theoretical peak**: With 4 FLOPs/cycle per core (AVX2+FMA), ~32 GFLOPS at 2GHz
   - **Practical limit**: Memory bandwidth becomes bottleneck for large k
   - **Optimization progression**: 1×1 → 1×4 → 4×4 → 8×8 → AVX2-optimized versions
4. **Architectural Considerations**:
   - **Register file size**: Determines maximum unrolling factor
   - **FMA units**: Modern CPUs can issue 2 FMA/cycle, requiring careful scheduling
   - **Load/store ports**: Balanced computation needs 2 loads per FMA operation

Educational Value

This implementation serves as a **critical pedagogical bridge** between: - **Naïve triple-loop** (no optimization) - **SIMD-vectorized kernels** (explicit vector instructions) - **Cache-blocked algorithms** (multi-level tiling)

It demonstrates that **even without explicit vector intrinsics**, careful loop structuring and unrolling can yield significant performance gains through better

register allocation and **instruction scheduling**.

Key Insight: This 4×4 blocking strategy achieves **moderate performance improvements** (typically 2-4× over baseline) but primarily serves as a **conceptual foundation** for more advanced optimizations that combine **register tiling**, **cache blocking**, and **SIMD vectorization** to approach theoretical hardware limits.

Source Code Implementation

```
/* Create macros so that the matrices are stored in column-major order */

#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Routine for computing C = A * B + C */

void AddDot4x4( int, double *, int, double *, int, double *, int );
void AddDot( int, double *, int, double *, double * );

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j;

  for ( j=0; j<n; j+=4 ){        /* Loop over the columns of C, unrolled by 4 */
    for ( i=0; i<m; i+=4 ){        /* Loop over the rows of C */
      /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
     one routine (four inner products) */

      AddDot4x4( k, &A( i,0 ), lda, &B( 0,j ), ldb, &C( i,j ), ldc );
    }
  }
}


void AddDot4x4( int k, double *a, int lda,  double *b, int ldb, double *c, int ldc )
{
  /* So, this routine computes a 4x4 block of matrix A

           C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ).
           C( 1, 0 ), C( 1, 1 ), C( 1, 2 ), C( 1, 3 ).
```

```
           C( 2, 0 ), C( 2, 1 ), C( 2, 2 ), C( 2, 3 ).
           C( 3, 0 ), C( 3, 1 ), C( 3, 2 ), C( 3, 3 ).

      Notice that this routine is called with c = C( i, j ) in the
      previous routine, so these are actually the elements

           C( i  , j ), C( i  , j+1 ), C( i  , j+2 ), C( i  , j+3 )
           C( i+1, j ), C( i+1, j+1 ), C( i+1, j+2 ), C( i+1, j+3 )
           C( i+2, j ), C( i+2, j+1 ), C( i+2, j+2 ), C( i+2, j+3 )
           C( i+3, j ), C( i+3, j+1 ), C( i+3, j+2 ), C( i+3, j+3 )

      in the original matrix C

      In this version, we merge each set of four loops, computing four
      inner products simultaneously. */

  int p;

  for ( p=0; p<k; p++ ){
    /* First row */
    C( 0, 0 ) += A( 0, p ) * B( p, 0 );
    C( 0, 1 ) += A( 0, p ) * B( p, 1 );
    C( 0, 2 ) += A( 0, p ) * B( p, 2 );
    C( 0, 3 ) += A( 0, p ) * B( p, 3 );

    /* Second row */
    C( 1, 0 ) += A( 1, p ) * B( p, 0 );
    C( 1, 1 ) += A( 1, p ) * B( p, 1 );
    C( 1, 2 ) += A( 1, p ) * B( p, 2 );
    C( 1, 3 ) += A( 1, p ) * B( p, 3 );

    /* Third row */
    C( 2, 0 ) += A( 2, p ) * B( p, 0 );
    C( 2, 1 ) += A( 2, p ) * B( p, 1 );
    C( 2, 2 ) += A( 2, p ) * B( p, 2 );
    C( 2, 3 ) += A( 2, p ) * B( p, 3 );

    /* Fourth row */
    C( 3, 0 ) += A( 3, p ) * B( p, 0 );
    C( 3, 1 ) += A( 3, p ) * B( p, 1 );
    C( 3, 2 ) += A( 3, p ) * B( p, 2 );
    C( 3, 3 ) += A( 3, p ) * B( p, 3 );
  }
}
```

File: MMult_4x4_6.c

Analysis of MMult_4x4_6.c: Register Blocking Optimization

1. Role & Purpose

This file implements a **matrix multiplication kernel** that computes $\mathbf{C} = \mathbf{A} \times \mathbf{B} + \mathbf{C}$ using **4×4 register blocking**. It serves as an educational example in a sequence of matrix multiplication optimizations, demonstrating how **manual register allocation** and **loop unrolling** can reduce memory traffic and improve performance. The code represents a crucial intermediate step between naive implementations and more advanced optimizations using **SIMD intrinsics** or **cache blocking**.

2. Technical Details

Memory Layout and Access Macros

```
#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]
```

- These macros implement **column-major ordering**, consistent with FORTRAN and BLAS conventions
- `lda`, `ldb`, `ldc` are **leading dimensions** (typically the matrix height) that enable submatrix addressing
- The indexing `(j)*lda + (i)` places consecutive rows `(i)` in contiguous memory for fixed column `(j)`

Main Loop Structure

```
void MY_MMult( int m, int n, int k, double *a, int lda,
               double *b, int ldb, double *c, int ldc )
{
  for ( j=0; j<n; j+=4 ){        /* Loop over columns of C (unrolled by 4) */
    for ( i=0; i<m; i+=4 ){      /* Loop over rows of C (unrolled by 4) */
      AddDot4x4( k, &A( i,0 ), lda, &B( 0,j ), ldb, &C( i,j ), ldc );
    }
  }
}
```

- **Outer loops** are unrolled by a factor of 4 in both dimensions
- Each iteration computes a **4×4 block** of matrix C
- The `AddDot4x4` function is called with pointers to:
  - `&A(i,0)`: Starting at row `i`, column 0 of A
  - `&B(0,j)`: Starting at row 0, column `j` of B

– `&C(i,j)`: Starting at row `i`, column `j` of C

Register Blocking Implementation

```
void AddDot4x4( int k, double *a, int lda, double *b, int ldb,
                double *c, int ldc )
{
  /* Register declarations for 4×4 C block */
  register double c_00_reg, c_01_reg, c_02_reg, c_03_reg,
                  c_10_reg, c_11_reg, c_12_reg, c_13_reg,
                  c_20_reg, c_21_reg, c_22_reg, c_23_reg,
                  c_30_reg, c_31_reg, c_32_reg, c_33_reg;

  /* Register declarations for 4 elements of A column */
  register double a_0p_reg, a_1p_reg, a_2p_reg, a_3p_reg;
```

Key Optimization Techniques: 1. **Register Allocation**: 16 accumulator registers for the C block, 4 registers for A elements 2. **Inner Loop Fusion**: The inner `p` loop computes all 16 dot products simultaneously 3. **Register Reuse**: Each A element (`a_*p_reg`) is reused 4 times against different B elements 4. **Accumulation in Registers**: Results accumulate in registers before final store to memory

Computation Pattern

For each iteration of `p` (inner dimension):

```
Load: a_0p_reg = A(0,p)  // Column p of A's 4×1 block
Load: B(p,0), B(p,1), B(p,2), B(p,3)  // Row p of B's 1×4 block

Compute:
c_00_reg += a_0p_reg * B(p,0)
c_01_reg += a_0p_reg * B(p,1)
c_02_reg += a_0p_reg * B(p,2)
c_03_reg += a_0p_reg * B(p,3)

Repeat for a_1p_reg, a_2p_reg, a_3p_reg
```

This implements the mathematical operation:
`C_block += A_block(:,p) × B_block(p,:)`

3. HPC Context

Performance Characteristics

- **Reduced Memory Traffic**: Each element of A is loaded once and used 4 times (against 4 columns of B)

- **Register Pressure**: Uses 20 double-precision registers (16 accumulators + 4 A elements)
- **Arithmetic Intensity**: 32 FLOPs (16 multiplies + 16 adds) per 8 loads (4 A + 4 B) = 4 FLOPs/load
- **Pipeline Utilization**: The unrolled structure exposes instruction-level parallelism

Architectural Considerations

1. **Register File Size**: Modern CPUs have 16-32 floating-point registers; this approach uses them aggressively
2. **Load/Store Unit**: Reduces store operations by accumulating in registers
3. **Instruction Cache**: Larger code size from unrolling may affect I-cache efficiency
4. **Compiler Hints**: The `register` keyword is advisory; modern compilers perform register allocation automatically

Limitations and Evolution

- **Fixed Block Size**: The 4×4 blocking may not be optimal for all architectures
- **No SIMD**: Does not use vector instructions (SSE/AVX)
- **No Cache Blocking**: Only register-level optimization; no L1/L2 cache optimization
- **Portability Issues**: `register` keyword is deprecated in C++17

Educational Value

This implementation demonstrates: - **Data reuse** through register blocking - **Loop unrolling** to reduce loop overhead - **Manual optimization** techniques that compilers may not automatically apply - The **foundation** for more advanced optimizations like: - SIMD vectorization (processing multiple elements per instruction) - Cache blocking (tiling for L1/L2 cache) - Loop reordering for better prefetching

Performance Scaling

For larger matrices, this kernel would typically be embedded within a **multi-level blocking strategy**: 1. **

Source Code Implementation

```
/* Create macros so that the matrices are stored in column-major order */

#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
```

```
#define C(i,j) c[ (j)*ldc + (i) ]

/* Routine for computing C = A * B + C */

void AddDot4x4( int, double *, int, double *, int, double *, int );

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j;

  for ( j=0; j<n; j+=4 ){        /* Loop over the columns of C, unrolled by 4 */
    for ( i=0; i<m; i+=4 ){        /* Loop over the rows of C */
      /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
     one routine (four inner products) */

      AddDot4x4( k, &A( i,0 ), lda, &B( 0,j ), ldb, &C( i,j ), ldc );
    }
  }
}


void AddDot4x4( int k, double *a, int lda,  double *b, int ldb, double *c, int ldc )
{
  /* So, this routine computes a 4x4 block of matrix A

           C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ).
           C( 1, 0 ), C( 1, 1 ), C( 1, 2 ), C( 1, 3 ).
           C( 2, 0 ), C( 2, 1 ), C( 2, 2 ), C( 2, 3 ).
           C( 3, 0 ), C( 3, 1 ), C( 3, 2 ), C( 3, 3 ).

     Notice that this routine is called with c = C( i, j ) in the
     previous routine, so these are actually the elements

           C( i  , j ), C( i  , j+1 ), C( i  , j+2 ), C( i  , j+3 )
           C( i+1, j ), C( i+1, j+1 ), C( i+1, j+2 ), C( i+1, j+3 )
           C( i+2, j ), C( i+2, j+1 ), C( i+2, j+2 ), C( i+2, j+3 )
           C( i+3, j ), C( i+3, j+1 ), C( i+3, j+2 ), C( i+3, j+3 )

     in the original matrix C

     In this version, we accumulate in registers and put A( 0, p ) in a register */

  int p;
  register double
```

```
  /* hold contributions to
     C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 )
     C( 1, 0 ), C( 1, 1 ), C( 1, 2 ), C( 1, 3 )
     C( 2, 0 ), C( 2, 1 ), C( 2, 2 ), C( 2, 3 )
     C( 3, 0 ), C( 3, 1 ), C( 3, 2 ), C( 3, 3 )   */
     c_00_reg,   c_01_reg,   c_02_reg,   c_03_reg,
     c_10_reg,   c_11_reg,   c_12_reg,   c_13_reg,
     c_20_reg,   c_21_reg,   c_22_reg,   c_23_reg,
     c_30_reg,   c_31_reg,   c_32_reg,   c_33_reg,
  /* hold
     A( 0, p )
     A( 1, p )
     A( 2, p )
     A( 3, p ) */
     a_0p_reg,
     a_1p_reg,
     a_2p_reg,
     a_3p_reg;

c_00_reg = 0.0;   c_01_reg = 0.0;   c_02_reg = 0.0;   c_03_reg = 0.0;
c_10_reg = 0.0;   c_11_reg = 0.0;   c_12_reg = 0.0;   c_13_reg = 0.0;
c_20_reg = 0.0;   c_21_reg = 0.0;   c_22_reg = 0.0;   c_23_reg = 0.0;
c_30_reg = 0.0;   c_31_reg = 0.0;   c_32_reg = 0.0;   c_33_reg = 0.0;

for ( p=0; p<k; p++ ){
  a_0p_reg = A( 0, p );
  a_1p_reg = A( 1, p );
  a_2p_reg = A( 2, p );
  a_3p_reg = A( 3, p );

  /* First row */
  c_00_reg += a_0p_reg * B( p, 0 );
  c_01_reg += a_0p_reg * B( p, 1 );
  c_02_reg += a_0p_reg * B( p, 2 );
  c_03_reg += a_0p_reg * B( p, 3 );

  /* Second row */
  c_10_reg += a_1p_reg * B( p, 0 );
  c_11_reg += a_1p_reg * B( p, 1 );
  c_12_reg += a_1p_reg * B( p, 2 );
  c_13_reg += a_1p_reg * B( p, 3 );

  /* Third row */
  c_20_reg += a_2p_reg * B( p, 0 );
  c_21_reg += a_2p_reg * B( p, 1 );
  c_22_reg += a_2p_reg * B( p, 2 );
```

```
    c_23_reg += a_2p_reg * B( p, 3 );

    /* Four row */
    c_30_reg += a_3p_reg * B( p, 0 );
    c_31_reg += a_3p_reg * B( p, 1 );
    c_32_reg += a_3p_reg * B( p, 2 );
    c_33_reg += a_3p_reg * B( p, 3 );
  }

  C( 0, 0 ) += c_00_reg;   C( 0, 1 ) += c_01_reg;   C( 0, 2 ) += c_02_reg;   C( 0, 3 ) += c_
  C( 1, 0 ) += c_10_reg;   C( 1, 1 ) += c_11_reg;   C( 1, 2 ) += c_12_reg;   C( 1, 3 ) += c_
  C( 2, 0 ) += c_20_reg;   C( 2, 1 ) += c_21_reg;   C( 2, 2 ) += c_22_reg;   C( 2, 3 ) += c_
  C( 3, 0 ) += c_30_reg;   C( 3, 1 ) += c_31_reg;   C( 3, 2 ) += c_32_reg;   C( 3, 3 ) += c_
}
```

File: MMult_4x4_7.c

Analysis of MMult_4x4_7.c

1. Role & Purpose

This file implements an **optimized matrix multiplication kernel** (`C = A × B + C`) using **4×4 loop unrolling** at both the outer loop and inner computation levels. Within the project infrastructure, it represents an intermediate optimization stage between naive triple-loop implementations and more advanced techniques using SIMD instructions or cache blocking. The code serves as:

- **A pedagogical example** demonstrating how **register blocking** and **pointer-based memory access** can improve performance
- **A foundation for further optimizations** by showing how to accumulate results in registers before writing back to memory
- **A reference implementation** for comparing against more sophisticated approaches like those using AVX/AVX2 intrinsics

The key innovation here is the **explicit management of 16 accumulator registers** for a 4×4 output block, combined with **pointer arithmetic** to traverse columns of matrix B efficiently.

2. Technical Details

Memory Access Pattern & Macros

```
#define A(i,j) a[ (j)*lda + (i) ]  // Column-major indexing
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]
```

These macros implement **column-major ordering**, where adjacent elements

in memory belong to the same column (differing row index `i`). This contrasts with row-major ordering and significantly impacts cache behavior.

Outer Loop Structure

```
for (j=0; j<n; j+=4) {      // Columns of C (unrolled by 4)
  for (i=0; i<m; i+=4) {    // Rows of C (unrolled by 4)
    AddDot4x4(...);
  }
}
```

The outer loops traverse the output matrix `C` in **4×4 blocks**, reducing loop overhead and enabling optimized inner kernel processing.

Inner Kernel: AddDot4x4 Register Allocation Strategy:

```
register double c_00_reg, c_01_reg, ..., c_33_reg;  // 16 accumulators
register double a_0p_reg, a_1p_reg, a_2p_reg, a_3p_reg;  // 4 A elements
double *b_p0_pntr, *b_p1_pntr, *b_p2_pntr, *b_p3_pntr;  // 4 B pointers
```

The `register` keyword (deprecated in C++ but meaningful in C) suggests to the compiler that these variables should be kept in CPU registers. The kernel maintains: - **16 accumulator registers** for the 4×4 block of C - **4 registers** for a column of 4 elements from matrix A - **4 pointers** to track positions in four consecutive columns of matrix B

Computation Pattern:

```
for (p=0; p<k; p++) {
  // Load 4 elements from column p of A
  a_0p_reg = A(0, p);
  a_1p_reg = A(1, p);
  a_2p_reg = A(2, p);
  a_3p_reg = A(3, p);

  // Set pointers to row p of B's 4 columns
  b_p0_pntr = &B(p, 0);
  b_p1_pntr = &B(p, 1);
  b_p2_pntr = &B(p, 2);
  b_p3_pntr = &B(p, 3);

  // Update all 16 accumulators
  c_00_reg += a_0p_reg * *b_p0_pntr;
  c_01_reg += a_0p_reg * *b_p1_pntr;
  // ... (14 more operations)
  c_33_reg += a_3p_reg * *b_p3_pntr++;
}
```

The inner loop performs **16 multiply-accumulate (FMA-like) operations per iteration** using a **single load of 4 A elements** reused across all 4 columns of B. The **post-increment on B pointers** (`*b_p0_pntr++`) advances each pointer to the next row in their respective columns.

Final Store Operation:

```
C(0, 0) += c_00_reg; C(0, 1) += c_01_reg; ...
```

The accumulated results are written back to memory only once per 4×4 block, minimizing **store instructions** and leveraging **register reuse**.

3. HPC Context

Performance Relevance

1. **Register Blocking & Temporal Locality:**
   - By keeping 16 accumulators in registers, the kernel achieves **zero register spilling** (assuming sufficient architectural registers)
   - The 4 A elements are reused 4 times each (once per B column), improving **arithmetic intensity**
   - This represents a **4×4 register tile** strategy that minimizes loads/stores relative to FLOPs
2. **Memory Access Patterns:**
   - **Column-major access to A**: When `p` increments, consecutive A elements (`A(0,p)`, `A(1,p)`, etc.) are contiguous in memory, enabling efficient cache lines
   - **Row-wise access to B**: Each B pointer traverses a row within its column. Since B is column-major, these accesses are **strided** (separated by `ldb`), which can cause cache inefficiencies
   - The pattern represents a **middle ground** between pure inner product and outer product formulations
3. **Instruction-Level Parallelism (ILP):**
   - The 16 independent multiply-accumulate operations within the loop body provide substantial **ILP opportunities** for out-of-order execution
   - Modern CPUs can potentially schedule many of these operations concurrently
4. **Limitations and Optimization Pathway:**
   - **No vectorization**: The scalar operations don't leverage SIMD units (SSE/AVX)
   - **No cache blocking**: The entire matrices are accessed in streaming fashion without reuse across 4×4 blocks
   - **Pointer chasing**: The four B pointers require separate load instructions; vector loads would be more efficient
   - This implementation typically achieves 20-40% of peak FLOPS on modern CPUs, serving as a baseline for **SIMD-optimized versions**

Educational Value This code exemplifies several key HPC concepts: - **Explicit register management** vs. relying on compiler optimization - **Loop unrolling** to reduce branch overhead - **Strength reduction** (pointer arithmetic instead of index computation) - **Data reuse** through multiple accumulators - The **trade-off** between complexity and performance

It directly prepares students for understanding: - **SIMD intrinsics** (replacing scalar operations with vector instructions) - **Cache-aware blocking** (adding outer loops to tile for L1/L2 cache) - **Assembly-level optimization** (seeing how high-level constructs map to machine instructions)

This implementation represents a crucial stepping stone toward production-level BLAS kernels, demonstrating that **manual optimization at the loop level** can yield significant performance gains before even considering architecture-specific features.

Source Code Implementation

```
/* Create macros so that the matrices are stored in column-major order */

#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Routine for computing C = A * B + C */

void AddDot4x4( int, double *, int, double *, int, double *, int );

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j;

  for ( j=0; j<n; j+=4 ){        /* Loop over the columns of C, unrolled by 4 */
    for ( i=0; i<m; i+=4 ){        /* Loop over the rows of C */
      /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
     one routine (four inner products) */

      AddDot4x4( k, &A( i,0 ), lda, &B( 0,j ), ldb, &C( i,j ), ldc );
    }
  }
}


void AddDot4x4( int k, double *a, int lda,  double *b, int ldb, double *c, int ldc )
```

```
{
  /* So, this routine computes a 4x4 block of matrix A

           C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ).
           C( 1, 0 ), C( 1, 1 ), C( 1, 2 ), C( 1, 3 ).
           C( 2, 0 ), C( 2, 1 ), C( 2, 2 ), C( 2, 3 ).
           C( 3, 0 ), C( 3, 1 ), C( 3, 2 ), C( 3, 3 ).

     Notice that this routine is called with c = C( i, j ) in the
     previous routine, so these are actually the elements

           C( i  , j ), C( i  , j+1 ), C( i  , j+2 ), C( i  , j+3 )
           C( i+1, j ), C( i+1, j+1 ), C( i+1, j+2 ), C( i+1, j+3 )
           C( i+2, j ), C( i+2, j+1 ), C( i+2, j+2 ), C( i+2, j+3 )
           C( i+3, j ), C( i+3, j+1 ), C( i+3, j+2 ), C( i+3, j+3 )

     in the original matrix C

     In this version, we use pointer to track where in four columns of B we are */

  int p;
  register double
    /* hold contributions to
       C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 )
       C( 1, 0 ), C( 1, 1 ), C( 1, 2 ), C( 1, 3 )
       C( 2, 0 ), C( 2, 1 ), C( 2, 2 ), C( 2, 3 )
       C( 3, 0 ), C( 3, 1 ), C( 3, 2 ), C( 3, 3 )   */
       c_00_reg,   c_01_reg,   c_02_reg,   c_03_reg,
       c_10_reg,   c_11_reg,   c_12_reg,   c_13_reg,
       c_20_reg,   c_21_reg,   c_22_reg,   c_23_reg,
       c_30_reg,   c_31_reg,   c_32_reg,   c_33_reg,
    /* hold
       A( 0, p )
       A( 1, p )
       A( 2, p )
       A( 3, p ) */
       a_0p_reg,
       a_1p_reg,
       a_2p_reg,
       a_3p_reg;
  double
    /* Point to the current elements in the four columns of B */
    *b_p0_pntr, *b_p1_pntr, *b_p2_pntr, *b_p3_pntr;

  c_00_reg = 0.0;   c_01_reg = 0.0;   c_02_reg = 0.0;   c_03_reg = 0.0;
  c_10_reg = 0.0;   c_11_reg = 0.0;   c_12_reg = 0.0;   c_13_reg = 0.0;
```

```
  c_20_reg = 0.0;   c_21_reg = 0.0;   c_22_reg = 0.0;   c_23_reg = 0.0;
  c_30_reg = 0.0;   c_31_reg = 0.0;   c_32_reg = 0.0;   c_33_reg = 0.0;

  for ( p=0; p<k; p++ ){
    a_0p_reg = A( 0, p );
    a_1p_reg = A( 1, p );
    a_2p_reg = A( 2, p );
    a_3p_reg = A( 3, p );

    b_p0_pntr = &B( p, 0 );
    b_p1_pntr = &B( p, 1 );
    b_p2_pntr = &B( p, 2 );
    b_p3_pntr = &B( p, 3 );

    /* First row */
    c_00_reg += a_0p_reg * *b_p0_pntr;
    c_01_reg += a_0p_reg * *b_p1_pntr;
    c_02_reg += a_0p_reg * *b_p2_pntr;
    c_03_reg += a_0p_reg * *b_p3_pntr;

    /* Second row */
    c_10_reg += a_1p_reg * *b_p0_pntr;
    c_11_reg += a_1p_reg * *b_p1_pntr;
    c_12_reg += a_1p_reg * *b_p2_pntr;
    c_13_reg += a_1p_reg * *b_p3_pntr;

    /* Third row */
    c_20_reg += a_2p_reg * *b_p0_pntr;
    c_21_reg += a_2p_reg * *b_p1_pntr;
    c_22_reg += a_2p_reg * *b_p2_pntr;
    c_23_reg += a_2p_reg * *b_p3_pntr;

    /* Four row */
    c_30_reg += a_3p_reg * *b_p0_pntr++;
    c_31_reg += a_3p_reg * *b_p1_pntr++;
    c_32_reg += a_3p_reg * *b_p2_pntr++;
    c_33_reg += a_3p_reg * *b_p3_pntr++;
  }

  C( 0, 0 ) += c_00_reg;   C( 0, 1 ) += c_01_reg;   C( 0, 2 ) += c_02_reg;   C( 0, 3 ) += c_
  C( 1, 0 ) += c_10_reg;   C( 1, 1 ) += c_11_reg;   C( 1, 2 ) += c_12_reg;   C( 1, 3 ) += c_
  C( 2, 0 ) += c_20_reg;   C( 2, 1 ) += c_21_reg;   C( 2, 2 ) += c_22_reg;   C( 2, 3 ) += c_
  C( 3, 0 ) += c_30_reg;   C( 3, 1 ) += c_31_reg;   C( 3, 2 ) += c_32_reg;   C( 3, 3 ) += c_
}
```

File: MMult_4x4_8.c

Analysis of `MMult_4x4_8.c`: Optimized Matrix Multiplication Kernel

1. Role & Purpose

This file implements a **highly optimized matrix multiplication kernel** designed for performance tuning in a pedagogical HPC context. It serves as an **intermediate optimization step** in a progression from naive to highly-tuned implementations, demonstrating key techniques for accelerating the fundamental $\mathbf{C} = \mathbf{A} \cdot \mathbf{B} + \mathbf{C}$ operation. The code specifically:

- Computes a **matrix-matrix multiplication** with 4×4 **blocking/unrolling** on both row and column dimensions
- Serves as a **benchmarking target** for understanding how **register allocation** and **loop unrolling** affect performance
- Provides a **foundation for further optimizations** like vectorization (AVX/SSE) and cache-aware blocking
- Demonstrates **manual optimization techniques** that compilers might not automatically apply

This implementation represents the **eighth stage** in an optimization series (as indicated by "_8" in the filename), focusing on **register reuse across multiple inner products**.

2. Technical Details

Macro Definitions for Column-Major Storage

```
#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]
```

These macros implement **column-major ordering**, consistent with Fortran and BLAS conventions. The `(j)*ld + (i)` indexing means: - `lda`, `ldb`, `ldc` are the **leading dimensions** (row counts including padding) - Elements in the same column are contiguous in memory (`i` changes fastest) - This affects **memory access patterns** and cache behavior significantly

Loop Structure in `MY_MMult`

```
for ( j=0; j<n; j+=4 ){        /* Loop over columns of C */
  for ( i=0; i<m; i+=4 ){      /* Loop over rows of C */
    AddDot4x4(...);
  }
}
```

The kernel uses **4×4 loop unrolling** on the output matrix C: - `j` loop strides by 4 columns (unrolled column-wise) - `i` loop strides by 4 rows (unrolled row-wise) - Each iteration computes a **4×4 block** of C - This reduces loop overhead and enables **register reuse** across multiple computations

Register Optimization in `AddDot4x4`

Register Variable Declarations

```
register double c_00_reg, c_01_reg, ..., a_0p_reg, ..., b_p0_reg, ...;
```

The use of `register` keyword (now largely advisory for compilers) indicates: - **16 C-element registers**: Four rows × four columns of partial results - **4 A-element registers**: One column slice (`a_0p_reg` to `a_3p_reg`) - **4 B-element registers**: One row slice (`b_p0_reg` to `b_p3_reg`) - **4 B pointers**: `b_p0_pntr` to `b_p3_pntr` for each column of B

Inner Loop Computation Pattern

```
for ( p=0; p<k; p++ ){
  a_0p_reg = A( 0, p );  // Load 4 elements from A
  a_1p_reg = A( 1, p );
  a_2p_reg = A( 2, p );
  a_3p_reg = A( 3, p );

  b_p0_reg = *b_p0_pntr++;  // Load 4 elements from B
  b_p1_reg = *b_p1_pntr++;
  b_p2_reg = *b_p2_pntr++;
  b_p3_reg = *b_p3_pntr++;

  // 16 multiply-add operations
  c_00_reg += a_0p_reg * b_p0_reg;
  c_01_reg += a_0p_reg * b_p1_reg;
  // ... (14 more FMA operations)
}
```

Key characteristics: - **Single load, multiple uses**: Each `a_Xp_reg` reused across 4 B columns - **Continuous pointer advancement**: B pointers increment through columns - **No memory traffic for C**: Accumulation happens entirely in registers - **16 FLOPs per iteration**: All using register-resident data

Memory Access Pattern

- **A access**: Strided by `lda` (column-major, loading 4×1 column slice)
- **B access**: Contiguous within each column (`b_pX_pntr++`)
- **C access**: Only at initialization and final store (16 stores total)

3. HPC Context & Performance Relevance

Memory Hierarchy Optimization

This implementation demonstrates **three crucial optimizations**:

1. **Register Blocking**: The 4×4 blocking keeps partial sums in registers throughout the inner loop, minimizing:
   - **Register pressure** by carefully allocating 24 register variables
   - **Register spilling** that would occur with larger blocks
   - **Memory traffic** to/from C matrix
2. **Spatial Locality Enhancement**:
   - B elements accessed contiguously within each column
   - A elements accessed with unit stride within each 4-element column slice
   - Enables **hardware prefetching** and **cache line efficiency**
3. **Temporal Locality Exploitation**:
   - Each loaded A element reused across 4 B columns
   - **Arithmetic intensity** increased to ~1.33 FLOPs/byte (16 FLOPs / 12 bytes loaded)
   - Reduces **memory bandwidth pressure**

Instruction-Level Parallelism (ILP)

- **16 independent multiply-add chains** per iteration
- Allows **out-of-order execution** and **pipeline filling**
- Limited only by functional unit availability and register dependencies

Limitations and Next Steps

While advanced for its stage, this implementation has **key limitations**:

1. **Fixed Block Size**: Hard-coded 4×4 blocks may not match:
   - **CPU register file size** (typically 16–32 FP registers)
   - **Cache line size** (typically 64 bytes = 8 doubles)
   - **Vector register width** (AVX: 4 doubles, AVX-512: 8 doubles)
2. **No Vectorization**: Manual **SIMD intrinsics** (AVX/SSE) could double/triple performance
3. **Cache Blocking Missing**: No **L1/L2 cache-aware tiling** for large matrices
4. **Alignment Ignored**: Unaligned memory access may cause penalties

Educational Value

This code exemplifies **the transition point** between: - **Scalar optimization** (register allocation, loop unrolling) - **Vector optimization** (next logical step:

AVX intrinsics) - **Cache optimization** (requires additional outer loop blocking)

It demonstrates **manual optimization principles** that remain relevant even with modern compilers, as automatic optimization often cannot achieve this level of **algorithmic restructuring**.

Performance Expectations

Compared to naive triple-loop implementation: - **~2–4× speedup** from reduced loop overhead and better register use - **Additional 2–4× potential** from vectorization (8–16× total) - **Order-of-magnitude gains** possible with full cache optimization

This implementation represents a **critical learning milestone** in understanding how to map matrix multiplication onto hardware resources efficiently, directly informing more advanced techniques used in libraries like **OpenBLAS**, **Intel MKL**, and **BLIS**.

Source Code Implementation

```
/* Create macros so that the matrices are stored in column-major order */

#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Routine for computing C = A * B + C */

void AddDot4x4( int, double *, int, double *, int, double *, int );

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j;

  for ( j=0; j<n; j+=4 ){        /* Loop over the columns of C, unrolled by 4 */
    for ( i=0; i<m; i+=4 ){        /* Loop over the rows of C */
      /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
     one routine (four inner products) */

      AddDot4x4( k, &A( i,0 ), lda, &B( 0,j ), ldb, &C( i,j ), ldc );
    }
  }
}
```

```
void AddDot4x4( int k, double *a, int lda,  double *b, int ldb, double *c, int ldc )
{
  /* So, this routine computes a 4x4 block of matrix A

           C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ).
           C( 1, 0 ), C( 1, 1 ), C( 1, 2 ), C( 1, 3 ).
           C( 2, 0 ), C( 2, 1 ), C( 2, 2 ), C( 2, 3 ).
           C( 3, 0 ), C( 3, 1 ), C( 3, 2 ), C( 3, 3 ).

     Notice that this routine is called with c = C( i, j ) in the
     previous routine, so these are actually the elements

           C( i  , j ), C( i  , j+1 ), C( i  , j+2 ), C( i  , j+3 )
           C( i+1, j ), C( i+1, j+1 ), C( i+1, j+2 ), C( i+1, j+3 )
           C( i+2, j ), C( i+2, j+1 ), C( i+2, j+2 ), C( i+2, j+3 )
           C( i+3, j ), C( i+3, j+1 ), C( i+3, j+2 ), C( i+3, j+3 )

     in the original matrix C

     In this version, we use registers for elements in the current row
     of B as well */

  int p;
  register double
    /* hold contributions to
       C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 )
       C( 1, 0 ), C( 1, 1 ), C( 1, 2 ), C( 1, 3 )
       C( 2, 0 ), C( 2, 1 ), C( 2, 2 ), C( 2, 3 )
       C( 3, 0 ), C( 3, 1 ), C( 3, 2 ), C( 3, 3 )   */
       c_00_reg,   c_01_reg,   c_02_reg,   c_03_reg,
       c_10_reg,   c_11_reg,   c_12_reg,   c_13_reg,
       c_20_reg,   c_21_reg,   c_22_reg,   c_23_reg,
       c_30_reg,   c_31_reg,   c_32_reg,   c_33_reg,
    /* hold
       A( 0, p )
       A( 1, p )
       A( 2, p )
       A( 3, p ) */
       a_0p_reg,
       a_1p_reg,
       a_2p_reg,
       a_3p_reg,
       b_p0_reg,
       b_p1_reg,
```

```
    b_p2_reg,
    b_p3_reg;

double
  /* Point to the current elements in the four columns of B */
  *b_p0_pntr, *b_p1_pntr, *b_p2_pntr, *b_p3_pntr;

b_p0_pntr = &B( 0, 0 );
b_p1_pntr = &B( 0, 1 );
b_p2_pntr = &B( 0, 2 );
b_p3_pntr = &B( 0, 3 );

c_00_reg = 0.0;   c_01_reg = 0.0;   c_02_reg = 0.0;   c_03_reg = 0.0;
c_10_reg = 0.0;   c_11_reg = 0.0;   c_12_reg = 0.0;   c_13_reg = 0.0;
c_20_reg = 0.0;   c_21_reg = 0.0;   c_22_reg = 0.0;   c_23_reg = 0.0;
c_30_reg = 0.0;   c_31_reg = 0.0;   c_32_reg = 0.0;   c_33_reg = 0.0;

for ( p=0; p<k; p++ ){
  a_0p_reg = A( 0, p );
  a_1p_reg = A( 1, p );
  a_2p_reg = A( 2, p );
  a_3p_reg = A( 3, p );

  b_p0_reg = *b_p0_pntr++;
  b_p1_reg = *b_p1_pntr++;
  b_p2_reg = *b_p2_pntr++;
  b_p3_reg = *b_p3_pntr++;

  /* First row */
  c_00_reg += a_0p_reg * b_p0_reg;
  c_01_reg += a_0p_reg * b_p1_reg;
  c_02_reg += a_0p_reg * b_p2_reg;
  c_03_reg += a_0p_reg * b_p3_reg;

  /* Second row */
  c_10_reg += a_1p_reg * b_p0_reg;
  c_11_reg += a_1p_reg * b_p1_reg;
  c_12_reg += a_1p_reg * b_p2_reg;
  c_13_reg += a_1p_reg * b_p3_reg;

  /* Third row */
  c_20_reg += a_2p_reg * b_p0_reg;
  c_21_reg += a_2p_reg * b_p1_reg;
  c_22_reg += a_2p_reg * b_p2_reg;
  c_23_reg += a_2p_reg * b_p3_reg;
```

```
    /* Four row */
    c_30_reg += a_3p_reg * b_p0_reg;
    c_31_reg += a_3p_reg * b_p1_reg;
    c_32_reg += a_3p_reg * b_p2_reg;
    c_33_reg += a_3p_reg * b_p3_reg;
  }

  C( 0, 0 ) += c_00_reg;   C( 0, 1 ) += c_01_reg;   C( 0, 2 ) += c_02_reg;   C( 0, 3 ) += c_
  C( 1, 0 ) += c_10_reg;   C( 1, 1 ) += c_11_reg;   C( 1, 2 ) += c_12_reg;   C( 1, 3 ) += c_
  C( 2, 0 ) += c_20_reg;   C( 2, 1 ) += c_21_reg;   C( 2, 2 ) += c_22_reg;   C( 2, 3 ) += c_
  C( 3, 0 ) += c_30_reg;   C( 3, 1 ) += c_31_reg;   C( 3, 2 ) += c_32_reg;   C( 3, 3 ) += c_
}
```

File: MMult_4x4_9.c

Analysis of MMult_4x4_9.c

1. Role & Purpose

This file implements a **blocked matrix multiplication kernel** designed to compute the operation $\mathbf{C} = \mathbf{A} \times \mathbf{B} + \mathbf{C}$ for matrices stored in **column-major order**. Within a pedagogical optimization framework, it represents an intermediate step focusing on **register-level blocking** and **loop unrolling**. The kernel breaks the computation into fixed 4×4 output blocks, optimizing data reuse and reducing loop overhead. This serves as a foundational example in a series of progressively optimized matrix multiplication implementations, illustrating how to structure computation for improved **data locality** and **instruction-level parallelism**.

2. Technical Details

Matrix Storage and Access Macros

- The macros `A(i,j)`, `B(i,j)`, and `C(i,j)` abstract column-major indexing, where `a[(j)*lda + (i)]` accesses element at row `i` and column `j`. This convention matches libraries like BLAS and Fortran, ensuring memory accesses follow column-wise strides, which can influence cache efficiency.

Outer Loop Structure in `MY_MMult`

- The function `MY_MMult` iterates over the output matrix **C** in 4×4 blocks. The outer loops step through columns (`j += 4`) and rows (`i += 4`), each time invoking `AddDot4x4` to compute a full 4×4 block of **C**.
- **Key optimization**: Loop unrolling by 4 in both dimensions reduces loop control overhead and enables the compiler to better schedule instructions.

Inner Kernel: `AddDot4x4`

- This function computes a 4×4 block of **C** by accumulating contributions from a **4×k** block of **A** and a **k×4** block of **B**. The inner loop over `p` (from 0 to `k-1`) performs a **rank-1 update** to the 4×4 block.

Register Variables and Pointer Arithmetic

- **Register variables**: The accumulators (`c_00_reg` to `c_33_reg`) and elements of **A** and **B** are declared with the `register` keyword, hinting to the compiler to keep these frequently accessed values in CPU registers rather than memory, minimizing latency.
- **Pointer arithmetic**: Four pointers (`b_p0_pntr` to `b_p3_pntr`) track the current positions in the four columns of **B** being processed. Incrementing these pointers (`*b_p0_pntr++`) efficiently strides through memory while reducing address computation overhead.

Computation Pattern

- The inner loop loads four values from **A** (`a_0p_reg` to `a_3p_reg`) and four from **B** (`b_p0_reg` to `b_p3_reg`), then updates all 16 accumulators with scalar multiplications and additions. This **explicit unrolling** exposes independent operations that can be executed in parallel by a superscalar processor.

Store Phase

- After the inner loop completes, the 16 register accumulators are written back to the corresponding locations in **C** using the column-major macro. This minimizes repeated memory accesses to **C** during accumulation.

3. HPC Context

Cache Efficiency Through Blocking

- By processing 4×4 blocks, the kernel exploits **temporal locality**: elements of **A** and **B** are reused multiple times within the inner loop. For example, each `a_0p_reg` is used in four multiply-add operations (with `b_p0_reg` to `b_p3_reg`), reducing the pressure on the memory hierarchy. This is a stepping stone toward larger cache-aware blockings (e.g., L1/L2 blocking).

Register Allocation and Instruction-Level Parallelism (ILP)

- The use of 16 separate register accumulators eliminates **write-after-read dependencies** on **C** during the inner loop, allowing out-of-order execution and pipelining. Modern CPUs with multiple functional units (e.g.,

FMA units) can theoretically issue several of these independent operations per cycle, though this scalar code does not yet exploit SIMD vectorization.

Reduction of Loop Overhead

- Unrolling the inner `p` loop (implicitly via the 4×4 structure) and the outer `i` and `j` loops reduces branch misprediction penalties and increases the ratio of floating-point operations to control instructions, improving **pipeline utilization**.

Foundation for Vectorization

- Although this implementation uses scalar operations, the structured 4×4 blocking and independent accumulators set the stage for **SIMD vectorization** (e.g., using AVX intrinsics). Each row of accumulators could be replaced with vector registers, allowing simultaneous computation of multiple columns.

Limitations and Trade-offs

- **Fixed block size**: The 4×4 blocking is hardcoded, which may not be optimal for all architectures or matrix sizes. Adaptive blocking based on cache sizes is often necessary for peak performance.
- **Scalar operations**: Without explicit vectorization, this code cannot fully utilize modern wide SIMD units. However, it demonstrates the critical step of restructuring computation to enable such optimizations.

This kernel exemplifies a **micro-kernel** approach, where small, highly optimized blocks form the building blocks for larger blocked algorithms. In practice, such kernels are often combined with multi-level blocking strategies to achieve high performance across varied architectures.

Source Code Implementation

```
/* Create macros so that the matrices are stored in column-major order */

#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Routine for computing C = A * B + C */

void AddDot4x4( int, double *, int, double *, int, double *, int );

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
```

```
                                    double *c, int ldc )
{
  int i, j;

  for ( j=0; j<n; j+=4 ){        /* Loop over the columns of C, unrolled by 4 */
    for ( i=0; i<m; i+=4 ){        /* Loop over the rows of C */
      /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
     one routine (four inner products) */

      AddDot4x4( k, &A( i,0 ), lda, &B( 0,j ), ldb, &C( i,j ), ldc );
    }
  }
}


void AddDot4x4( int k, double *a, int lda,  double *b, int ldb, double *c, int ldc )
{
  /* So, this routine computes a 4x4 block of matrix A

           C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ).
           C( 1, 0 ), C( 1, 1 ), C( 1, 2 ), C( 1, 3 ).
           C( 2, 0 ), C( 2, 1 ), C( 2, 2 ), C( 2, 3 ).
           C( 3, 0 ), C( 3, 1 ), C( 3, 2 ), C( 3, 3 ).

     Notice that this routine is called with c = C( i, j ) in the
     previous routine, so these are actually the elements

           C( i  , j ), C( i  , j+1 ), C( i  , j+2 ), C( i  , j+3 )
           C( i+1, j ), C( i+1, j+1 ), C( i+1, j+2 ), C( i+1, j+3 )
           C( i+2, j ), C( i+2, j+1 ), C( i+2, j+2 ), C( i+2, j+3 )
           C( i+3, j ), C( i+3, j+1 ), C( i+3, j+2 ), C( i+3, j+3 )

     in the original matrix C

     A simple rearrangement to prepare for the use of vector registers */

  int p;
  register double
    /* hold contributions to
       C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 )
       C( 1, 0 ), C( 1, 1 ), C( 1, 2 ), C( 1, 3 )
       C( 2, 0 ), C( 2, 1 ), C( 2, 2 ), C( 2, 3 )
       C( 3, 0 ), C( 3, 1 ), C( 3, 2 ), C( 3, 3 )   */
      c_00_reg,   c_01_reg,   c_02_reg,   c_03_reg,
      c_10_reg,   c_11_reg,   c_12_reg,   c_13_reg,
      c_20_reg,   c_21_reg,   c_22_reg,   c_23_reg,
```

```
    c_30_reg,   c_31_reg,   c_32_reg,   c_33_reg,
  /* hold
     A( 0, p )
     A( 1, p )
     A( 2, p )
     A( 3, p ) */
     a_0p_reg,
     a_1p_reg,
     a_2p_reg,
     a_3p_reg,
     b_p0_reg,
     b_p1_reg,
     b_p2_reg,
     b_p3_reg;

double
  /* Point to the current elements in the four columns of B */
  *b_p0_pntr, *b_p1_pntr, *b_p2_pntr, *b_p3_pntr;

b_p0_pntr = &B( 0, 0 );
b_p1_pntr = &B( 0, 1 );
b_p2_pntr = &B( 0, 2 );
b_p3_pntr = &B( 0, 3 );

c_00_reg = 0.0;   c_01_reg = 0.0;   c_02_reg = 0.0;   c_03_reg = 0.0;
c_10_reg = 0.0;   c_11_reg = 0.0;   c_12_reg = 0.0;   c_13_reg = 0.0;
c_20_reg = 0.0;   c_21_reg = 0.0;   c_22_reg = 0.0;   c_23_reg = 0.0;
c_30_reg = 0.0;   c_31_reg = 0.0;   c_32_reg = 0.0;   c_33_reg = 0.0;

for ( p=0; p<k; p++ ){
  a_0p_reg = A( 0, p );
  a_1p_reg = A( 1, p );
  a_2p_reg = A( 2, p );
  a_3p_reg = A( 3, p );

  b_p0_reg = *b_p0_pntr++;
  b_p1_reg = *b_p1_pntr++;
  b_p2_reg = *b_p2_pntr++;
  b_p3_reg = *b_p3_pntr++;

  /* First row and second rows */
  c_00_reg += a_0p_reg * b_p0_reg;
  c_10_reg += a_1p_reg * b_p0_reg;

  c_01_reg += a_0p_reg * b_p1_reg;
  c_11_reg += a_1p_reg * b_p1_reg;
```

```
    c_02_reg += a_0p_reg * b_p2_reg;
    c_12_reg += a_1p_reg * b_p2_reg;

    c_03_reg += a_0p_reg * b_p3_reg;
    c_13_reg += a_1p_reg * b_p3_reg;

    /* Third and fourth rows */
    c_20_reg += a_2p_reg * b_p0_reg;
    c_30_reg += a_3p_reg * b_p0_reg;

    c_21_reg += a_2p_reg * b_p1_reg;
    c_31_reg += a_3p_reg * b_p1_reg;

    c_22_reg += a_2p_reg * b_p2_reg;
    c_32_reg += a_3p_reg * b_p2_reg;

    c_23_reg += a_2p_reg * b_p3_reg;
    c_33_reg += a_3p_reg * b_p3_reg;
  }

  C( 0, 0 ) += c_00_reg;   C( 0, 1 ) += c_01_reg;   C( 0, 2 ) += c_02_reg;   C( 0, 3 ) += c_
  C( 1, 0 ) += c_10_reg;   C( 1, 1 ) += c_11_reg;   C( 1, 2 ) += c_12_reg;   C( 1, 3 ) += c_
  C( 2, 0 ) += c_20_reg;   C( 2, 1 ) += c_21_reg;   C( 2, 2 ) += c_22_reg;   C( 2, 3 ) += c_
  C( 3, 0 ) += c_30_reg;   C( 3, 1 ) += c_31_reg;   C( 3, 2 ) += c_32_reg;   C( 3, 3 ) += c_
}
```

Chapter 6: Memory Opt (Packing)

File: MMult_4x4_11.c

Analysis of MMult_4x4_11.c

1. Role & Purpose

This file implements an **optimized matrix multiplication kernel** within a performance exploration framework. Its primary function is to compute the matrix product $\mathbf{C} = \mathbf{A} \times \mathbf{B} + \mathbf{C}$ using a sophisticated multi-level optimization strategy that combines **cache-aware blocking**, **loop unrolling**, and **SIMD vectorization** using SSE intrinsics. The code serves as an educational demonstration of how to systematically apply HPC optimization techniques to achieve near-peak floating-point performance on modern CPU architectures.

2. Technical Details

Macro Definitions and Memory Layout

```
#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]
```

- **Column-major ordering**: The macros implement Fortran-style storage where consecutive elements in memory belong to the same column. This affects memory access patterns and cache behavior.
- **Leading dimension parameters** (`lda`, `ldb`, `ldc`): Allow handling of submatrices within larger matrices, essential for blocked algorithms.

Three-Level Optimization Hierarchy Level 1: Outer Blocking (Cache Optimization)

```
#define mc 256  // Block size for rows of C/A
#define kc 128  // Block size for reduction dimension

for (p=0; p<k; p+=kc) {
    pb = min(k-p, kc);
    for (i=0; i<m; i+=mc) {
        ib = min(m-i, mc);
        InnerKernel(ib, n, pb, &A(i,p), lda, &B(p,0), ldb, &C(i,0), ldc);
    }
}
```

- **`kc` blocking**: Partitions the reduction dimension (k) to keep the `pb × n` block of B in L2/L3 cache.
- **`mc` blocking**: Partitions rows of A/C to keep the `ib × pb` block of A in L1 cache.
- This **two-level blocking** minimizes cache misses by ensuring working sets fit in appropriate cache levels.

Level 2: Inner Kernel (Register Blocking)

```
for (j=0; j<n; j+=4) {
    for (i=0; i<m; i+=4) {
        AddDot4x4(k, &A(i,0), lda, &B(0,j), ldb, &C(i,j), ldc);
    }
}
```

- **4×4 register blocking**: Processes 4 rows and 4 columns of C simultaneously.
- Each iteration computes 16 dot products, amortizing loop overhead and enabling effective vectorization.
- The fixed block size allows complete unrolling and register allocation.

Level 3: Micro-Kernel with SIMD Vectorization

```c
typedef union {
    __m128d v;
    double d[2];
} v2df_t;
```

- **SSE vectorization**: Uses 128-bit vector registers (`__m128d`) holding 2 double-precision values.
- **Union type**: Enables both vector operations and scalar access to individual elements.

Vectorized Computation Pattern

```c
a_0p_a_1p_vreg.v = _mm_load_pd((double *) &A(0, p));  // Load A[0,p] and A[1,p]
b_p0_vreg.v = _mm_loaddup_pd((double *) b_p0_pntr++); // Broadcast B[p,0]

c_00_c_10_vreg.v += a_0p_a_1p_vreg.v * b_p0_vreg.v;   // Fused multiply-add
```

1. **Vector loads**: `_mm_load_pd()` loads two consecutive elements from matrix A (rows 0 and 1).
2. **Broadcast loads**: `_mm_loaddup_pd()` loads and duplicates a single B element across both vector lanes.
3. **Fused multiply-add**: Each vector operation computes two partial dot products simultaneously.
4. **Register accumulation**: Eight vector registers hold 16 accumulator values (4×4 block).

Efficient Memory Access Patterns

- **Column-wise B access**: Four pointers (`b_p0_pntr` to `b_p3_pntr`) stream through consecutive columns of B.
- **Row-wise A access**: Two vector loads fetch four rows of A's current column.
- This pattern achieves **unit stride access** for both matrices, maximizing cache line utilization.

3. HPC Context

Cache Hierarchy Optimization

- **`kc` dimension blocking**: The 128×n block of B remains in L2/L3 cache across multiple iterations of the i-loop, avoiding capacity misses.
- **`mc` dimension blocking**: The 256×128 block of A fits in L1 cache, ensuring fast access during the inner kernel execution.
- **Register blocking**: The 4×4 micro-kernel keeps 16 accumulator values in vector registers, eliminating loads/stores to the innermost loop.

SIMD Parallelism Exploitation

- **2-way double-precision SIMD**: SSE instructions process two operations per cycle, doubling theoretical peak performance.
- **Instruction-level parallelism**: Multiple independent vector operations (8 FMAs per iteration) can be pipelined in modern superscalar processors.
- **Reduced instruction count**: Vector instructions replace multiple scalar operations, reducing front-end pressure.

Memory Bandwidth Optimization

- **Data reuse**: Each element of A is used 4 times (for 4 columns of C), each element of B is used 4 times (for 4 rows of C).
- **Cache line utilization**: Unit stride accesses ensure full utilization of 64-byte cache lines.
- **Prefetching friendly**: Regular access patterns enable hardware prefetchers to operate effectively.

Arithmetic Intensity Balance For the 4×4 micro-kernel processing k elements: - **Operations**: 16 × (2k) = 32k flops (16 dot products, each with k multiply-adds) - **Data movement**: (4 × k) + (4 × k) = 8k elements loaded - **Arithmetic intensity**: 4 flops/element, well-balanced for modern processors

Limitations and Trade-offs

- **Fixed block sizes**: Optimal values depend on specific cache sizes (would benefit from auto-tuning).
- **SSE2 limitation**: AVX/AVX-512 could provide 4×/8× better vectorization.
- **No multithreading**: Pure single-threaded implementation missing additional parallelism.
- **Alignment assumptions**: `_mm_load_pd()` requires 16-byte alignment not explicitly guaranteed.

This implementation demonstrates a **classical optimization progression** from naive triple-loop to cache-blocked, register-blocked, and vectorized computation—a foundational case study in HPC algorithm design.

Source Code Implementation

```
/* Create macros so that the matrices are stored in column-major order */

#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Block sizes */
#define mc 256
```

```
#define kc 128

#define min( i, j ) ( (i)<(j) ? (i): (j) )

/* Routine for computing C = A * B + C */

void AddDot4x4( int, double *, int, double *, int, double *, int );

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, j, p, pb, ib;

  /* This time, we compute a mc x n block of C by a call to the InnerKernel */

  for ( p=0; p<k; p+=kc ){
    pb = min( k-p, kc );
    for ( i=0; i<m; i+=mc ){
      ib = min( m-i, mc );
      InnerKernel( ib, n, pb, &A( i,p ), lda, &B(p, 0 ), ldb, &C( i,0 ), ldc );
    }
  }
}

void InnerKernel( int m, int n, int k, double *a, int lda,
                                       double *b, int ldb,
                                       double *c, int ldc )
{
  int i, j;

  for ( j=0; j<n; j+=4 ){        /* Loop over the columns of C, unrolled by 4 */
    for ( i=0; i<m; i+=4 ){        /* Loop over the rows of C */
      /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
     one routine (four inner products) */

      AddDot4x4( k, &A( i,0 ), lda, &B( 0,j ), ldb, &C( i,j ), ldc );
    }
  }
}

#include <mmintrin.h>
#include <xmmintrin.h>  // SSE
#include <pmmintrin.h>  // SSE2
#include <emmintrin.h>  // SSE3
```

```
typedef union
{
  __m128d v;
  double d[2];
} v2df_t;

void AddDot4x4( int k, double *a, int lda,  double *b, int ldb, double *c, int ldc )
{
  /* So, this routine computes a 4x4 block of matrix A

           C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ).
           C( 1, 0 ), C( 1, 1 ), C( 1, 2 ), C( 1, 3 ).
           C( 2, 0 ), C( 2, 1 ), C( 2, 2 ), C( 2, 3 ).
           C( 3, 0 ), C( 3, 1 ), C( 3, 2 ), C( 3, 3 ).

     Notice that this routine is called with c = C( i, j ) in the
     previous routine, so these are actually the elements

           C( i  , j ), C( i  , j+1 ), C( i  , j+2 ), C( i  , j+3 )
           C( i+1, j ), C( i+1, j+1 ), C( i+1, j+2 ), C( i+1, j+3 )
           C( i+2, j ), C( i+2, j+1 ), C( i+2, j+2 ), C( i+2, j+3 )
           C( i+3, j ), C( i+3, j+1 ), C( i+3, j+2 ), C( i+3, j+3 )

     in the original matrix C

     And now we use vector registers and instructions */

  int p;
  v2df_t
    c_00_c_10_vreg,    c_01_c_11_vreg,    c_02_c_12_vreg,    c_03_c_13_vreg,
    c_20_c_30_vreg,    c_21_c_31_vreg,    c_22_c_32_vreg,    c_23_c_33_vreg,
    a_0p_a_1p_vreg,
    a_2p_a_3p_vreg,
    b_p0_vreg, b_p1_vreg, b_p2_vreg, b_p3_vreg;

  double
    /* Point to the current elements in the four columns of B */
    *b_p0_pntr, *b_p1_pntr, *b_p2_pntr, *b_p3_pntr;

  b_p0_pntr = &B( 0, 0 );
  b_p1_pntr = &B( 0, 1 );
  b_p2_pntr = &B( 0, 2 );
  b_p3_pntr = &B( 0, 3 );

  c_00_c_10_vreg.v = _mm_setzero_pd();
  c_01_c_11_vreg.v = _mm_setzero_pd();
```

```
  c_02_c_12_vreg.v = _mm_setzero_pd();
  c_03_c_13_vreg.v = _mm_setzero_pd();
  c_20_c_30_vreg.v = _mm_setzero_pd();
  c_21_c_31_vreg.v = _mm_setzero_pd();
  c_22_c_32_vreg.v = _mm_setzero_pd();
  c_23_c_33_vreg.v = _mm_setzero_pd();

  for ( p=0; p<k; p++ ){
    a_0p_a_1p_vreg.v = _mm_load_pd( (double *) &A( 0, p ) );
    a_2p_a_3p_vreg.v = _mm_load_pd( (double *) &A( 2, p ) );

    b_p0_vreg.v = _mm_loaddup_pd( (double *) b_p0_pntr++ );   /* load and duplicate */
    b_p1_vreg.v = _mm_loaddup_pd( (double *) b_p1_pntr++ );   /* load and duplicate */
    b_p2_vreg.v = _mm_loaddup_pd( (double *) b_p2_pntr++ );   /* load and duplicate */
    b_p3_vreg.v = _mm_loaddup_pd( (double *) b_p3_pntr++ );   /* load and duplicate */

    /* First row and second rows */
    c_00_c_10_vreg.v += a_0p_a_1p_vreg.v * b_p0_vreg.v;
    c_01_c_11_vreg.v += a_0p_a_1p_vreg.v * b_p1_vreg.v;
    c_02_c_12_vreg.v += a_0p_a_1p_vreg.v * b_p2_vreg.v;
    c_03_c_13_vreg.v += a_0p_a_1p_vreg.v * b_p3_vreg.v;

    /* Third and fourth rows */
    c_20_c_30_vreg.v += a_2p_a_3p_vreg.v * b_p0_vreg.v;
    c_21_c_31_vreg.v += a_2p_a_3p_vreg.v * b_p1_vreg.v;
    c_22_c_32_vreg.v += a_2p_a_3p_vreg.v * b_p2_vreg.v;
    c_23_c_33_vreg.v += a_2p_a_3p_vreg.v * b_p3_vreg.v;
  }

  C( 0, 0 ) += c_00_c_10_vreg.d[0];  C( 0, 1 ) += c_01_c_11_vreg.d[0];
  C( 0, 2 ) += c_02_c_12_vreg.d[0];  C( 0, 3 ) += c_03_c_13_vreg.d[0];

  C( 1, 0 ) += c_00_c_10_vreg.d[1];  C( 1, 1 ) += c_01_c_11_vreg.d[1];
  C( 1, 2 ) += c_02_c_12_vreg.d[1];  C( 1, 3 ) += c_03_c_13_vreg.d[1];

  C( 2, 0 ) += c_20_c_30_vreg.d[0];  C( 2, 1 ) += c_21_c_31_vreg.d[0];
  C( 2, 2 ) += c_22_c_32_vreg.d[0];  C( 2, 3 ) += c_23_c_33_vreg.d[0];

  C( 3, 0 ) += c_20_c_30_vreg.d[1];  C( 3, 1 ) += c_21_c_31_vreg.d[1];
  C( 3, 2 ) += c_22_c_32_vreg.d[1];  C( 3, 3 ) += c_23_c_33_vreg.d[1];
}
```

File: MMult_4x4_12.c

Analysis of `MMult_4x4_12.c`

1. Role & Purpose

This file implements **highly optimized double-precision matrix multiplication** (`C = A·B + C`) using a multi-layered optimization strategy. It serves as a **key pedagogical example** demonstrating how to combine: - **Blocking/tiling** for cache efficiency - **Data packing** for contiguous memory access - **SIMD vectorization** using SSE intrinsics - **Register blocking** to minimize memory operations

The implementation specifically targets **column-major storage** (common in Fortran and BLAS/LAPACK) and employs a **4×4 micro-kernel** as its computational heart, making it a **foundational example** in the progression from naive to highly optimized matrix multiplication.

---

2. Technical Details

Macro System & Memory Layout

```
#define A(i,j) a[ (j)*lda + (i) ]  // Column-major indexing
```

- **Column-major order**: Elements of each column are stored contiguously in memory
- **Leading dimension (lda)**: Allows working with submatrices of larger matrices
- **Macro abstraction**: Simplifies code readability while maintaining performance

Blocking Strategy

```
#define mc 256  // Block size for m dimension
#define kc 128  // Block size for k dimension
```

- **Two-level blocking**:
  1. **Outer blocking** (`MY_MMult`): Partitions computation into `mc × n × kc` blocks
  2. **Inner blocking** (`InnerKernel`): Processes 4×4 sub-blocks within each outer block
- **Block size selection**: `mc=256`, `kc=128` chosen to fit typical L2/L3 cache sizes

Packing Transformation

```
void PackMatrixA( int k, double *a, int lda, double *a_to )
```

- **Purpose**: Converts non-contiguous rows of A into contiguous, packed format
- **Operation**: For each column of A, copies 4 adjacent elements into consecutive memory
- **Benefit**: Enables aligned SIMD loads (`_mm_load_pd`) and improves spatial locality

Vectorized Micro-Kernel

```
void AddDot4x4( int k, double *a, int lda, double *b, int ldb, double *c, int ldc )
```

Key SIMD components: - **Vector type**: `v2df_t` union allows access as `__m128d` (SIMD) or `double[2]` (scalar) - **SIMD operations**: - `_mm_setzero_pd()`: Initialize vector to zero - `_mm_load_pd()`: Load two packed doubles (requires 16-byte alignment) - `_mm_loaddup_pd()`: Load and duplicate scalar across both vector lanes - Vector multiply-add: `c += a * b` (fused multiply-add pattern)

Register blocking strategy: - **C matrix**: 8 vector registers hold entire 4×4 block (2 rows per register) - **A matrix**: 2 vector registers hold current two rows of packed A - **B matrix**: 4 vector registers hold broadcasted elements from 4 columns - **Kernel unrolling**: Inner loop processes one iteration of `p` with maximal data reuse

Computation pattern for one iteration:

```
// Process two rows of A with four columns of B simultaneously
c_00_c_10_vreg += a_0p_a_1p_vreg * b_p0_vreg;  // Rows 0,1 × Col 0
c_01_c_11_vreg += a_0p_a_1p_vreg * b_p1_vreg;  // Rows 0,1 × Col 1
// ... similar for remaining columns
```

Execution Flow

1. **Outer loop** (`MY_MMult`): Tiles along `k` dimension with `kc` blocks
2. **Middle loop**: Tiles along `m` dimension with `mc` blocks
3. **InnerKernel**: Processes `mc × n` block using 4×4 micro-kernel
4. **Micro-kernel**: Vectorized computation of 4×4 blocks with packed A

---

3. HPC Context

Cache Hierarchy Optimization

- **L2/L3 cache blocking**: `mc × kc` block of A (256×128 = 256KB) fits in L3 cache
- **L1 cache optimization**: 4×4 micro-kernel with packed A ensures all working data fits in registers/L1
- **Data reuse**: Each packed A element reused across 4 columns of B (4× reuse)

SIMD Vectorization Efficiency

- **Theoretical peak**: 2 double operations per cycle per SSE2 unit
- **Actual achieved**: 8 multiply-add operations per iteration (4×2) using 2 vector units
- **Broadcast optimization**: `_mm_loaddup_pd()` minimizes B matrix bandwidth

Memory Access Patterns

- **Streaming access**: Packed A allows contiguous, aligned loads
- **Column-major consistency**: B accessed with stride-1 in inner loop (crucial for performance)
- **Register pressure**: 14 vector registers used (close to x86's 16-register limit)

Performance Implications

1. **Reduced cache misses**: Blocking minimizes capacity misses in outer loops
2. **High FLOP/byte ratio**: Register blocking reduces memory operations
3. **SIMD utilization**: 100% vectorization in micro-kernel
4. **Instruction-level parallelism**: Multiple independent multiply-add chains

Limitations & Trade-offs

- **Fixed block size**: Assumes matrices are multiples of 4 (requires edge handling in practice)
- **SSE2 only**: Modern systems could use AVX/AVX-512 for wider vectors
- **Power-of-two assumptions**: Block sizes chosen as powers of two for alignment
- **Column-major bias**: Performance degrades if row-major data must be converted

Educational Value This implementation demonstrates the **critical progression** in HPC optimization: 1. Naive triple loop → 2. Loop reordering → 3. Blocking → 4. Packing → 5. Vectorization It serves as a **bridge** between algorithmic understanding (cache-aware blocking) and hardware exploitation (SIMD programming), making it an **essential case study** in computational linear algebra optimization.

Key Insight: This code exemplifies the **"block-pack-vectorize"** paradigm that underpins modern high-performance linear algebra libraries like OpenBLAS and Intel MKL, showing how careful layering of optimizations can achieve orders-of-magnitude speedup over naive implementations.

Source Code Implementation

```
/* Create macros so that the matrices are stored in column-major order */

#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Block sizes */
#define mc 256
#define kc 128

#define min( i, j ) ( (i)<(j) ? (i): (j) )

/* Routine for computing C = A * B + C */

void AddDot4x4( int, double *, int, double *, int, double *, int );
void PackMatrixA( int, double *, int, double * );

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, p, pb, ib;

  /* This time, we compute a mc x n block of C by a call to the InnerKernel */

  for ( p=0; p<k; p+=kc ){
    pb = min( k-p, kc );
    for ( i=0; i<m; i+=mc ){
      ib = min( m-i, mc );
      InnerKernel( ib, n, pb, &A( i,p ), lda, &B(p, 0 ), ldb, &C( i,0 ), ldc );
    }
  }
}

void InnerKernel( int m, int n, int k, double *a, int lda,
                                       double *b, int ldb,
                                       double *c, int ldc )
{
  int i, j;
  double
    packedA[ m * k ];

  for ( j=0; j<n; j+=4 ){        /* Loop over the columns of C, unrolled by 4 */
```

```
    for ( i=0; i<m; i+=4 ){        /* Loop over the rows of C */
      /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
     one routine (four inner products) */
      PackMatrixA( k, &A( i, 0 ), lda, &packedA[ i*k ] );
      AddDot4x4( k, &packedA[ i*k ], 4, &B( 0,j ), ldb, &C( i,j ), ldc );
    }
  }
}

void PackMatrixA( int k, double *a, int lda, double *a_to )
{
  int j;

  for( j=0; j<k; j++){  /* loop over columns of A */
    double
      *a_ij_pntr = &A( 0, j );

    *a_to++ = *a_ij_pntr;
    *a_to++ = *(a_ij_pntr+1);
    *a_to++ = *(a_ij_pntr+2);
    *a_to++ = *(a_ij_pntr+3);
  }
}

#include <mmintrin.h>
#include <xmmintrin.h>  // SSE
#include <pmmintrin.h>  // SSE2
#include <emmintrin.h>  // SSE3

typedef union
{
  __m128d v;
  double d[2];
} v2df_t;

void AddDot4x4( int k, double *a, int lda,  double *b, int ldb, double *c, int ldc )
{
  /* So, this routine computes a 4x4 block of matrix A

           C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ).
           C( 1, 0 ), C( 1, 1 ), C( 1, 2 ), C( 1, 3 ).
           C( 2, 0 ), C( 2, 1 ), C( 2, 2 ), C( 2, 3 ).
           C( 3, 0 ), C( 3, 1 ), C( 3, 2 ), C( 3, 3 ).

     Notice that this routine is called with c = C( i, j ) in the
     previous routine, so these are actually the elements
```

```
          C( i  , j ), C( i  , j+1 ), C( i  , j+2 ), C( i  , j+3 )
          C( i+1, j ), C( i+1, j+1 ), C( i+1, j+2 ), C( i+1, j+3 )
          C( i+2, j ), C( i+2, j+1 ), C( i+2, j+2 ), C( i+2, j+3 )
          C( i+3, j ), C( i+3, j+1 ), C( i+3, j+2 ), C( i+3, j+3 )

   in the original matrix C

   And now we use vector registers and instructions */

int p;
v2df_t
  c_00_c_10_vreg,    c_01_c_11_vreg,    c_02_c_12_vreg,    c_03_c_13_vreg,
  c_20_c_30_vreg,    c_21_c_31_vreg,    c_22_c_32_vreg,    c_23_c_33_vreg,
  a_0p_a_1p_vreg,
  a_2p_a_3p_vreg,
  b_p0_vreg, b_p1_vreg, b_p2_vreg, b_p3_vreg;

double
  /* Point to the current elements in the four columns of B */
  *b_p0_pntr, *b_p1_pntr, *b_p2_pntr, *b_p3_pntr;

b_p0_pntr = &B( 0, 0 );
b_p1_pntr = &B( 0, 1 );
b_p2_pntr = &B( 0, 2 );
b_p3_pntr = &B( 0, 3 );

c_00_c_10_vreg.v = _mm_setzero_pd();
c_01_c_11_vreg.v = _mm_setzero_pd();
c_02_c_12_vreg.v = _mm_setzero_pd();
c_03_c_13_vreg.v = _mm_setzero_pd();
c_20_c_30_vreg.v = _mm_setzero_pd();
c_21_c_31_vreg.v = _mm_setzero_pd();
c_22_c_32_vreg.v = _mm_setzero_pd();
c_23_c_33_vreg.v = _mm_setzero_pd();

for ( p=0; p<k; p++ ){
  a_0p_a_1p_vreg.v = _mm_load_pd( (double *) &A( 0, p ) );
  a_2p_a_3p_vreg.v = _mm_load_pd( (double *) &A( 2, p ) );

  b_p0_vreg.v = _mm_loaddup_pd( (double *) b_p0_pntr++ );   /* load and duplicate */
  b_p1_vreg.v = _mm_loaddup_pd( (double *) b_p1_pntr++ );   /* load and duplicate */
  b_p2_vreg.v = _mm_loaddup_pd( (double *) b_p2_pntr++ );   /* load and duplicate */
  b_p3_vreg.v = _mm_loaddup_pd( (double *) b_p3_pntr++ );   /* load and duplicate */

  /* First row and second rows */
```

```
    c_00_c_10_vreg.v += a_0p_a_1p_vreg.v * b_p0_vreg.v;
    c_01_c_11_vreg.v += a_0p_a_1p_vreg.v * b_p1_vreg.v;
    c_02_c_12_vreg.v += a_0p_a_1p_vreg.v * b_p2_vreg.v;
    c_03_c_13_vreg.v += a_0p_a_1p_vreg.v * b_p3_vreg.v;

    /* Third and fourth rows */
    c_20_c_30_vreg.v += a_2p_a_3p_vreg.v * b_p0_vreg.v;
    c_21_c_31_vreg.v += a_2p_a_3p_vreg.v * b_p1_vreg.v;
    c_22_c_32_vreg.v += a_2p_a_3p_vreg.v * b_p2_vreg.v;
    c_23_c_33_vreg.v += a_2p_a_3p_vreg.v * b_p3_vreg.v;
  }

  C( 0, 0 ) += c_00_c_10_vreg.d[0];  C( 0, 1 ) += c_01_c_11_vreg.d[0];
  C( 0, 2 ) += c_02_c_12_vreg.d[0];  C( 0, 3 ) += c_03_c_13_vreg.d[0];

  C( 1, 0 ) += c_00_c_10_vreg.d[1];  C( 1, 1 ) += c_01_c_11_vreg.d[1];
  C( 1, 2 ) += c_02_c_12_vreg.d[1];  C( 1, 3 ) += c_03_c_13_vreg.d[1];

  C( 2, 0 ) += c_20_c_30_vreg.d[0];  C( 2, 1 ) += c_21_c_31_vreg.d[0];
  C( 2, 2 ) += c_22_c_32_vreg.d[0];  C( 2, 3 ) += c_23_c_33_vreg.d[0];

  C( 3, 0 ) += c_20_c_30_vreg.d[1];  C( 3, 1 ) += c_21_c_31_vreg.d[1];
  C( 3, 2 ) += c_22_c_32_vreg.d[1];  C( 3, 3 ) += c_23_c_33_vreg.d[1];
}
```

File: MMult_4x4_13.c

Analysis of MMult_4x4_13.c

1. Role & Purpose

This file implements a **highly optimized matrix multiplication kernel** (`C = A × B + C`) using a multi-level optimization strategy. It serves as an **advanced educational example** demonstrating how to combine:

- **Blocking/tiling** for cache hierarchy optimization
- **Data packing/prefetching** to improve memory access patterns
- **SIMD vectorization** using SSE intrinsics for parallel floating-point operations
- **Register blocking** to maximize data reuse in registers

The code represents a **production-quality optimization level** similar to techniques used in high-performance linear algebra libraries like BLAS (specifically, the GotoBLAS/OpenBLAS approach).

2. Technical Details

Macro System & Storage Order

```
#define A(i,j) a[ (j)*lda + (i) ]
```

These macros implement **column-major ordering** (consistent with Fortran and LAPACK conventions). Element `A(i,j)` is at memory location `a + j*lda + i`, where `lda` is the **leading dimension** (typically the number of rows).

Multi-Level Blocking Strategy Level 1: Outer Kernel (`MY_MMult`)

```
for (p=0; p<k; p+=kc) {
  for (i=0; i<m; i+=mc) {
    InnerKernel(ib, n, pb, ...);
  }
}
```

- **kc (128)**: Block size in the K-dimension (inner product dimension)
- **mc (256)**: Block size in the M-dimension (rows of C/A)
- The code processes the multiplication in **blocks** to keep working sets in cache

Level 2: Inner Kernel (`InnerKernel`) Processes a `mc × n` block of C using packed data:

```
for (j=0; j<n; j+=4) {      // Unroll columns by 4
  for (i=0; i<m; i+=4) {   // Process 4x4 micro-blocks
    if (j == 0) PackMatrixA(...);
    AddDot4x4(...);
  }
}
```

- Uses **4×4 micro-kernel** for register-level optimization
- **Packs matrix A** once per row block reuse

Data Packing (`PackMatrixA`)

```
for(j=0; j<k; j++) {
  *a_to++ = *a_ij_pntr;
  *a_to++ = *(a_ij_pntr+1);
  *a_to++ = *(a_ij_pntr+2);
  *a_to++ = *(a_ij_pntr+3);
}
```

- Converts **column-major 4×k block** of A into **contiguous packed format**
- Enables **unit-stride access** during computation (critical for vectorization)
- Reduces cache conflict misses by removing large strides

SIMD Vectorization with SSE (`AddDot4x4`) Data Types & Organization:

```
typedef union {
  __m128d v;      // SSE vector (2 doubles)
  double d[2];    // Scalar access
} v2df_t;
```

Uses **SSE2/SSE3** intrinsics for double-precision arithmetic.

Key Optimization Techniques:

1. **Register Blocking**: 12 vector registers hold:
   - 8 accumulate C values (4 registers × 2 rows each)
   - 2 for packed A columns
   - 4 for broadcast B elements
2. **Broadcast Loading**:

```
b_p0_vreg.v = _mm_loaddup_pd((double *) b_p0_pntr++);
```

`_mm_loaddup_pd()` loads and duplicates a scalar to both lanes of a vector, enabling **Fused Multiply-Add (FMA)** pattern.

3. **Vector Accumulation**:

```
c_00_c_10_vreg.v += a_0p_a_1p_vreg.v * b_p0_vreg.v;
```

Each instruction computes **2 multiply-add operations** in parallel.

4. **Memory Access Pattern**:

- Packed A: **unit-stride contiguous access** (perfect for vector loads)
- B: **broadcast single elements** to vector registers
- C: **scattered write-back** after accumulation

Execution Flow

```
MY_MMult (L3 cache blocking)
  ↓
InnerKernel (L2/L1 cache blocking)
  ↓
PackMatrixA (memory layout transformation)
  ↓
AddDot4x4 (register blocking + SIMD)
```

3. HPC Context

Cache Hierarchy Optimization

- **kc = 128**: Fits packed A block (4×128×8 = 4KB) in **L1 cache**
- **mc = 256**: Allows C block (256×n×8) to reside in **L2 cache**

- **Block reuse**: Packed A reused across multiple B columns (amortizes packing cost)

Memory Access Efficiency

- **Eliminates TLB misses**: Packed arrays use contiguous allocation
- **Reduces cache conflicts**: Removes large power-of-two strides
- **Enables prefetching**: Regular access patterns help hardware prefetchers

SIMD Utilization

- **Theoretical peak**: 2 FLOPs/cycle per SSE vector (2 doubles)
- **Achieved performance**: High due to:
  - **Register blocking** minimizes L1 accesses
  - **Balanced pipeline**: 2 load ports + 1 SIMD unit in typical x86
  - **Instruction-level parallelism**: Multiple independent vector operations

Arithmetic Intensity For 4×4 micro-kernel: - **Operations**: 4×4×2 = 32 FLOPs per inner iteration - **Data movement**: 4×2 A elements + 4 B elements = 12 elements - **Byte/FLOP**: (12×8)/32 = 3 bytes/FLOP (close to machine balance point)

Limitations & Trade-offs

1. **Fixed micro-kernel size**: 4×4 optimal for SSE (2-wide), not for AVX/AVX-512
2. **Packing overhead**: ~10-20% overhead for small matrices
3. **Alignment requirements**: `_mm_load_pd()` requires 16-byte alignment
4. **Register pressure**: 12/16 SSE registers used, limiting unrolling

Performance Characteristics

- **Expected performance**: 80-90% of theoretical peak on Haswell+ (with FMA)
- **Scalability**: Limited to single core; requires OpenMP for multi-core
- **Portability**: x86-specific; needs different intrinsics for ARM/GPU

This implementation demonstrates the **critical optimization path** for dense linear algebra: blocking for cache → packing for stride-1 access → vectorization for SIMD → register blocking for reuse. It forms the foundation upon which modern libraries like BLIS and OpenBLAS build their portable high-performance kernels.

Source Code Implementation

```
/* Create macros so that the matrices are stored in column-major order */
```

```
#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Block sizes */
#define mc 256
#define kc 128

#define min( i, j ) ( (i)<(j) ? (i): (j) )

/* Routine for computing C = A * B + C */

void AddDot4x4( int, double *, int, double *, int, double *, int );
void PackMatrixA( int, double *, int, double * );

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, p, pb, ib;

  /* This time, we compute a mc x n block of C by a call to the InnerKernel */

  for ( p=0; p<k; p+=kc ){
    pb = min( k-p, kc );
    for ( i=0; i<m; i+=mc ){
      ib = min( m-i, mc );
      InnerKernel( ib, n, pb, &A( i,p ), lda, &B(p, 0 ), ldb, &C( i,0 ), ldc );
    }
  }
}

void InnerKernel( int m, int n, int k, double *a, int lda,
                                       double *b, int ldb,
                                       double *c, int ldc )
{
  int i, j;
  double
    packedA[ m * k ];

  for ( j=0; j<n; j+=4 ){        /* Loop over the columns of C, unrolled by 4 */
    for ( i=0; i<m; i+=4 ){        /* Loop over the rows of C */
      /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
     one routine (four inner products) */
      if ( j == 0 ) PackMatrixA( k, &A( i, 0 ), lda, &packedA[ i*k ] );
```

```
      AddDot4x4( k, &packedA[ i*k ], 4, &B( 0,j ), ldb, &C( i,j ), ldc );
    }
  }
}

void PackMatrixA( int k, double *a, int lda, double *a_to )
{
  int j;

  for( j=0; j<k; j++){  /* loop over columns of A */
    double
      *a_ij_pntr = &A( 0, j );

    *a_to++ = *a_ij_pntr;
    *a_to++ = *(a_ij_pntr+1);
    *a_to++ = *(a_ij_pntr+2);
    *a_to++ = *(a_ij_pntr+3);
  }
}

#include <mmintrin.h>
#include <xmmintrin.h>  // SSE
#include <pmmintrin.h>  // SSE2
#include <emmintrin.h>  // SSE3

typedef union
{
  __m128d v;
  double d[2];
} v2df_t;

void AddDot4x4( int k, double *a, int lda,  double *b, int ldb, double *c, int ldc )
{
  /* So, this routine computes a 4x4 block of matrix A

           C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ).
           C( 1, 0 ), C( 1, 1 ), C( 1, 2 ), C( 1, 3 ).
           C( 2, 0 ), C( 2, 1 ), C( 2, 2 ), C( 2, 3 ).
           C( 3, 0 ), C( 3, 1 ), C( 3, 2 ), C( 3, 3 ).

     Notice that this routine is called with c = C( i, j ) in the
     previous routine, so these are actually the elements

           C( i  , j ), C( i  , j+1 ), C( i  , j+2 ), C( i  , j+3 )
           C( i+1, j ), C( i+1, j+1 ), C( i+1, j+2 ), C( i+1, j+3 )
           C( i+2, j ), C( i+2, j+1 ), C( i+2, j+2 ), C( i+2, j+3 )
```

```
        C( i+3, j ), C( i+3, j+1 ), C( i+3, j+2 ), C( i+3, j+3 )

   in the original matrix C

   And now we use vector registers and instructions */

int p;
v2df_t
  c_00_c_10_vreg,    c_01_c_11_vreg,    c_02_c_12_vreg,    c_03_c_13_vreg,
  c_20_c_30_vreg,    c_21_c_31_vreg,    c_22_c_32_vreg,    c_23_c_33_vreg,
  a_0p_a_1p_vreg,
  a_2p_a_3p_vreg,
  b_p0_vreg, b_p1_vreg, b_p2_vreg, b_p3_vreg;

double
  /* Point to the current elements in the four columns of B */
  *b_p0_pntr, *b_p1_pntr, *b_p2_pntr, *b_p3_pntr;

b_p0_pntr = &B( 0, 0 );
b_p1_pntr = &B( 0, 1 );
b_p2_pntr = &B( 0, 2 );
b_p3_pntr = &B( 0, 3 );

c_00_c_10_vreg.v = _mm_setzero_pd();
c_01_c_11_vreg.v = _mm_setzero_pd();
c_02_c_12_vreg.v = _mm_setzero_pd();
c_03_c_13_vreg.v = _mm_setzero_pd();
c_20_c_30_vreg.v = _mm_setzero_pd();
c_21_c_31_vreg.v = _mm_setzero_pd();
c_22_c_32_vreg.v = _mm_setzero_pd();
c_23_c_33_vreg.v = _mm_setzero_pd();

for ( p=0; p<k; p++ ){
  a_0p_a_1p_vreg.v = _mm_load_pd( (double *) a );
  a_2p_a_3p_vreg.v = _mm_load_pd( (double *) ( a+2 ) );
  a += 4;

  b_p0_vreg.v = _mm_loaddup_pd( (double *) b_p0_pntr++ );   /* load and duplicate */
  b_p1_vreg.v = _mm_loaddup_pd( (double *) b_p1_pntr++ );   /* load and duplicate */
  b_p2_vreg.v = _mm_loaddup_pd( (double *) b_p2_pntr++ );   /* load and duplicate */
  b_p3_vreg.v = _mm_loaddup_pd( (double *) b_p3_pntr++ );   /* load and duplicate */

  /* First row and second rows */
  c_00_c_10_vreg.v += a_0p_a_1p_vreg.v * b_p0_vreg.v;
  c_01_c_11_vreg.v += a_0p_a_1p_vreg.v * b_p1_vreg.v;
  c_02_c_12_vreg.v += a_0p_a_1p_vreg.v * b_p2_vreg.v;
```

```
    c_03_c_13_vreg.v += a_0p_a_1p_vreg.v * b_p3_vreg.v;

    /* Third and fourth rows */
    c_20_c_30_vreg.v += a_2p_a_3p_vreg.v * b_p0_vreg.v;
    c_21_c_31_vreg.v += a_2p_a_3p_vreg.v * b_p1_vreg.v;
    c_22_c_32_vreg.v += a_2p_a_3p_vreg.v * b_p2_vreg.v;
    c_23_c_33_vreg.v += a_2p_a_3p_vreg.v * b_p3_vreg.v;
  }

  C( 0, 0 ) += c_00_c_10_vreg.d[0];  C( 0, 1 ) += c_01_c_11_vreg.d[0];
  C( 0, 2 ) += c_02_c_12_vreg.d[0];  C( 0, 3 ) += c_03_c_13_vreg.d[0];

  C( 1, 0 ) += c_00_c_10_vreg.d[1];  C( 1, 1 ) += c_01_c_11_vreg.d[1];
  C( 1, 2 ) += c_02_c_12_vreg.d[1];  C( 1, 3 ) += c_03_c_13_vreg.d[1];

  C( 2, 0 ) += c_20_c_30_vreg.d[0];  C( 2, 1 ) += c_21_c_31_vreg.d[0];
  C( 2, 2 ) += c_22_c_32_vreg.d[0];  C( 2, 3 ) += c_23_c_33_vreg.d[0];

  C( 3, 0 ) += c_20_c_30_vreg.d[1];  C( 3, 1 ) += c_21_c_31_vreg.d[1];
  C( 3, 2 ) += c_22_c_32_vreg.d[1];  C( 3, 3 ) += c_23_c_33_vreg.d[1];
}
```

File: MMult_4x4_14.c

Analysis of MMult_4x4_14.c - An Optimized Matrix Multiplication Kernel

1. Role & Purpose

This file implements **an optimized double-precision matrix multiplication kernel** (`C = A × B + C`) designed for **high-performance computing** on x86 architectures. It serves as an advanced implementation demonstrating:

- **Memory hierarchy optimization** through multi-level blocking
- **SIMD vectorization** using SSE/SSE2 intrinsics for 4×4 micro-kernels
- **Data packing** to improve cache locality
- **Column-major storage** convention (consistent with BLAS/LAPACK)

The code represents **Level 3 BLAS** functionality (matrix-matrix operations) and targets modern CPUs with **vector processing capabilities**. It's designed as a production-grade kernel that would form the computational core of linear algebra libraries.

2. Technical Details

Macro System & Memory Layout

```
#define A(i,j) a[ (j)*lda + (i) ]
```

- **Column-major ordering**: Element `(i,j)` is stored at offset `j*lda + i`
- **Leading dimension** (`lda`, `ldb`, `ldc`): Distance between consecutive elements in memory along the column direction
- **Pointer arithmetic**: Enables natural indexing while maintaining performance

Multi-Level Blocking Strategy **Level 1: Outer blocking** (in `MY_MMult`):

```
#define mc 256  /* Block size for rows of C/A */
#define kc 128  /* Block size for inner dimension */
```

- **mc × kc blocks of A** and **kc × n blocks of B** fit in L2 cache
- Minimizes main memory accesses by reusing cached data

Level 2: Inner kernel blocking (in `InnerKernel`): - Operates on **4×4 sub-blocks** using vector registers - **Register blocking**: 8 vector registers hold intermediate C values - Enables **maximum data reuse** within CPU registers

Packing Mechanism **PackMatrixA** (packing columns of A): - Converts **4×k block** of A to **contiguous memory layout** - Stores each 4-element column contiguously (4 doubles) - Enables **vectorized loads** via `_mm_load_pd()`

PackMatrixB (packing columns of B): - Packs **k×4 block** of B with **stride removal** - Reorganizes for efficient **broadcast operations** - Uses **four separate pointers** for each column

Vectorized Micro-Kernel (`AddDot4x4`) **SIMD Architecture Utilization**:

```
#include <emmintrin.h>  // SSE2 intrinsics
typedef union { __m128d v; double d[2]; } v2df_t;
```

- **__m128d**: 128-bit vector holding 2 double-precision values
- **Union type**: Enables both vector operations and scalar access

Register Allocation Strategy:

```
v2df_t c_00_c_10_vreg, c_01_c_11_vreg, ...;  /* 8 accumulator registers */
```

- **8 vector registers** store partial results for 4×4 C block
- Each register holds **two consecutive row elements** from same column
- Example: `c_00_c_10_vreg` stores `C(0,j)` and `C(1,j)`

Vectorized Computation Loop:

```
for ( p=0; p<k; p++ ) {
    a_0p_a_1p_vreg.v = _mm_load_pd( (double *) a );  /* Load 2 rows of A */
    b_p0_vreg.v = _mm_loaddup_pd( (double *) b );    /* Broadcast B element */
```

```
    c_00_c_10_vreg.v += a_0p_a_1p_vreg.v * b_p0_vreg.v;
}
```

- `_mm_load_pd()`: Loads 2 contiguous doubles (aligned)
- `_mm_loaddup_pd()`: Loads and duplicates scalar to both vector lanes
- **Fused multiply-add**: Vectorized `a * b` accumulation

Memory Access Pattern:

```
A accesses: Stride-4 loads (packed columns)
B accesses: 4 broadcast operations per iteration
C accesses: Only initial zeroing and final store
```

3. HPC Context & Performance Relevance

Cache Hierarchy Optimization

- **L2 Cache Blocking**: `mc × kc = 256 × 128 = 32KB` (fits L2 cache)
- **L1 Cache/Load Unit**: 4×4 blocks fit in **vector registers**
- **Data packing** eliminates cache thrashing by ensuring:
  - **Spatial locality**: Contiguous access patterns
  - **Temporal locality**: Reuse within register block

Computational Intensity

- **Register-level blocking**: 4×4 block performs **16 flops per iteration**
- **Vectorization**: 2 operations per SIMD instruction → **8× speedup** over scalar
- **Arithmetic Intensity**: `flops/bytes` ratio maximized by:
  - Loading 4 A elements → reused across 4 B columns
  - Loading 4 B elements → reused across 4 A rows

Pipeline Efficiency

- **Loop unrolling**: 4× unrolling reduces branch overhead
- **Independent operations**: 8 vector accumulators enable **instruction-level parallelism**
- **Memory/compute overlap**: Packing decouples data movement from computation

Architectural Considerations

- **SSE2 Instruction Set**: Available on all x86-64 processors
- **Register pressure**: 8 vector registers + 4 temporary registers = **12/16 SSE registers used**
- **Memory alignment**: `_mm_load_pd()` requires 16-byte alignment (assumed by packing)

Performance Characteristics

- **Theoretical peak**: 2 doubles × 2 FMA × frequency × cores
- **Practical limit**: Memory bandwidth for large matrices
- **Optimal regime**: Block sizes tuned for typical cache sizes (L1: 32KB, L2: 256KB)

Scalability Considerations

- **Fixed 4×4 blocking**: Limits to SSE2; AVX/AVX-512 would use 8×8 or 16×16 blocks
- **Single-threaded**: Would need OpenMP pragmas for multi-core
- **Static blocking**: Could benefit from auto-tuning for different architectures

This implementation represents a **classical optimization approach** that balances computational intensity, data locality, and vectorization—a foundation for modern libraries like OpenBLAS and Intel MKL. The techniques demonstrated remain relevant for **any memory-bound operation** on hierarchical memory systems.

Source Code Implementation

```
/* Create macros so that the matrices are stored in column-major order */

#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Block sizes */
#define mc 256
#define kc 128

#define min( i, j ) ( (i)<(j) ? (i): (j) )

/* Routine for computing C = A * B + C */

void AddDot4x4( int, double *, int, double *, int, double *, int );
void PackMatrixA( int, double *, int, double * );
void PackMatrixB( int, double *, int, double * );
void InnerKernel( int, int, int, double *, int, double *, int, double *, int, int );

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, p, pb, ib;
```

```
  /* This time, we compute a mc x n block of C by a call to the InnerKernel */

  for ( p=0; p<k; p+=kc ){
    pb = min( k-p, kc );
    for ( i=0; i<m; i+=mc ){
      ib = min( m-i, mc );
      InnerKernel( ib, n, pb, &A( i,p ), lda, &B(p, 0 ), ldb, &C( i,0 ), ldc, i==0 );
    }
  }
}

void InnerKernel( int m, int n, int k, double *a, int lda,
                                       double *b, int ldb,
                                       double *c, int ldc, int first_time )
{
  int i, j;
  double
    packedA[ m * k ], packedB[ k*n ];

  for ( j=0; j<n; j+=4 ){        /* Loop over the columns of C, unrolled by 4 */
    PackMatrixB( k, &B( 0, j ), ldb, &packedB[ j*k ] );
    for ( i=0; i<m; i+=4 ){        /* Loop over the rows of C */
      /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
     one routine (four inner products) */
      if ( j == 0 )
    PackMatrixA( k, &A( i, 0 ), lda, &packedA[ i*k ] );
      AddDot4x4( k, &packedA[ i*k ], 4, &packedB[ j*k ], k, &C( i,j ), ldc );
    }
  }
}

void PackMatrixA( int k, double *a, int lda, double *a_to )
{
  int j;

  for( j=0; j<k; j++){  /* loop over columns of A */
    double
      *a_ij_pntr = &A( 0, j );

    *a_to     = *a_ij_pntr;
    *(a_to+1) = *(a_ij_pntr+1);
    *(a_to+2) = *(a_ij_pntr+2);
    *(a_to+3) = *(a_ij_pntr+3);

    a_to += 4;
```

```
  }
}

void PackMatrixB( int k, double *b, int ldb, double *b_to )
{
  int i;
  double
    *b_i0_pntr = &B( 0, 0 ), *b_i1_pntr = &B( 0, 1 ),
    *b_i2_pntr = &B( 0, 2 ), *b_i3_pntr = &B( 0, 3 );

  for( i=0; i<k; i++){  /* loop over rows of B */
    *b_to++ = *b_i0_pntr++;
    *b_to++ = *b_i1_pntr++;
    *b_to++ = *b_i2_pntr++;
    *b_to++ = *b_i3_pntr++;
  }
}

#include <mmintrin.h>
#include <xmmintrin.h>  // SSE
#include <pmmintrin.h>  // SSE2
#include <emmintrin.h>  // SSE3

typedef union
{
  __m128d v;
  double d[2];
} v2df_t;

void AddDot4x4( int k, double *a, int lda,  double *b, int ldb, double *c, int ldc )
{
  /* So, this routine computes a 4x4 block of matrix A

           C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ).
           C( 1, 0 ), C( 1, 1 ), C( 1, 2 ), C( 1, 3 ).
           C( 2, 0 ), C( 2, 1 ), C( 2, 2 ), C( 2, 3 ).
           C( 3, 0 ), C( 3, 1 ), C( 3, 2 ), C( 3, 3 ).

     Notice that this routine is called with c = C( i, j ) in the
     previous routine, so these are actually the elements

           C( i  , j ), C( i  , j+1 ), C( i  , j+2 ), C( i  , j+3 )
           C( i+1, j ), C( i+1, j+1 ), C( i+1, j+2 ), C( i+1, j+3 )
           C( i+2, j ), C( i+2, j+1 ), C( i+2, j+2 ), C( i+2, j+3 )
           C( i+3, j ), C( i+3, j+1 ), C( i+3, j+2 ), C( i+3, j+3 )
```

```
   in the original matrix C

   And now we use vector registers and instructions */

int p;
v2df_t
  c_00_c_10_vreg,    c_01_c_11_vreg,    c_02_c_12_vreg,    c_03_c_13_vreg,
  c_20_c_30_vreg,    c_21_c_31_vreg,    c_22_c_32_vreg,    c_23_c_33_vreg,
  a_0p_a_1p_vreg,
  a_2p_a_3p_vreg,
  b_p0_vreg, b_p1_vreg, b_p2_vreg, b_p3_vreg;

c_00_c_10_vreg.v = _mm_setzero_pd();
c_01_c_11_vreg.v = _mm_setzero_pd();
c_02_c_12_vreg.v = _mm_setzero_pd();
c_03_c_13_vreg.v = _mm_setzero_pd();
c_20_c_30_vreg.v = _mm_setzero_pd();
c_21_c_31_vreg.v = _mm_setzero_pd();
c_22_c_32_vreg.v = _mm_setzero_pd();
c_23_c_33_vreg.v = _mm_setzero_pd();

for ( p=0; p<k; p++ ){
  a_0p_a_1p_vreg.v = _mm_load_pd( (double *) a );
  a_2p_a_3p_vreg.v = _mm_load_pd( (double *) ( a+2 ) );
  a += 4;

  b_p0_vreg.v = _mm_loaddup_pd( (double *) b );       /* load and duplicate */
  b_p1_vreg.v = _mm_loaddup_pd( (double *) (b+1) );   /* load and duplicate */
  b_p2_vreg.v = _mm_loaddup_pd( (double *) (b+2) );   /* load and duplicate */
  b_p3_vreg.v = _mm_loaddup_pd( (double *) (b+3) );   /* load and duplicate */

  b += 4;

  /* First row and second rows */
  c_00_c_10_vreg.v += a_0p_a_1p_vreg.v * b_p0_vreg.v;
  c_01_c_11_vreg.v += a_0p_a_1p_vreg.v * b_p1_vreg.v;
  c_02_c_12_vreg.v += a_0p_a_1p_vreg.v * b_p2_vreg.v;
  c_03_c_13_vreg.v += a_0p_a_1p_vreg.v * b_p3_vreg.v;

  /* Third and fourth rows */
  c_20_c_30_vreg.v += a_2p_a_3p_vreg.v * b_p0_vreg.v;
  c_21_c_31_vreg.v += a_2p_a_3p_vreg.v * b_p1_vreg.v;
  c_22_c_32_vreg.v += a_2p_a_3p_vreg.v * b_p2_vreg.v;
  c_23_c_33_vreg.v += a_2p_a_3p_vreg.v * b_p3_vreg.v;
}
```

```
  C( 0, 0 ) += c_00_c_10_vreg.d[0];  C( 0, 1 ) += c_01_c_11_vreg.d[0];
  C( 0, 2 ) += c_02_c_12_vreg.d[0];  C( 0, 3 ) += c_03_c_13_vreg.d[0];

  C( 1, 0 ) += c_00_c_10_vreg.d[1];  C( 1, 1 ) += c_01_c_11_vreg.d[1];
  C( 1, 2 ) += c_02_c_12_vreg.d[1];  C( 1, 3 ) += c_03_c_13_vreg.d[1];

  C( 2, 0 ) += c_20_c_30_vreg.d[0];  C( 2, 1 ) += c_21_c_31_vreg.d[0];
  C( 2, 2 ) += c_22_c_32_vreg.d[0];  C( 2, 3 ) += c_23_c_33_vreg.d[0];

  C( 3, 0 ) += c_20_c_30_vreg.d[1];  C( 3, 1 ) += c_21_c_31_vreg.d[1];
  C( 3, 2 ) += c_22_c_32_vreg.d[1];  C( 3, 3 ) += c_23_c_33_vreg.d[1];
}
```

File: MMult_4x4_15.c

Analysis of `MMult_4x4_15.c`

1. Role & Purpose

This file implements **a highly optimized matrix multiplication kernel** using a multi-level blocking strategy combined with **SIMD vectorization**. It represents an advanced stage in the optimization pipeline where:

- **Cache-aware blocking** is implemented via parameters `mc`, `kc`, and `nb`
- **Register blocking** uses a 4×4 micro-kernel with SSE2/SSE3 intrinsics
- **Explicit data packing** transforms matrices into cache-friendly layouts
- **Memory hierarchy optimization** minimizes cache misses through careful data reuse patterns

The code computes `C = A×B + C` where all matrices are stored in **column-major order**, making it compatible with Fortran/LAPACK conventions. This is production-level code that demonstrates how theoretical HPC concepts translate to practical implementation.

2. Technical Details

Matrix Storage and Access Macros

```
#define A(i,j) a[ (j)*lda + (i) ]
```

The macros implement **column-major addressing**: element `(i,j)` is at position `j*lda + i`, where `lda` is the leading dimension (typically equals the number of rows). This differs from row-major (C-style) where it would be `i*lda + j`.

Three-Level Blocking Strategy The blocking parameters create a hierarchical decomposition: - **`mc` (256)**: Number of rows of C computed in one outer block - **`kc` (128)**: Number of columns of A (rows of B) processed in middle

blocks - **nb (1000)**: Number of columns of C computed (though not fully utilized in this kernel)

The blocking in `MY_MMult` follows:

```
for (p=0; p<k; p+=kc) {           // Loop over kc-sized panels of A and B
    pb = min(k-p, kc);            // Actual panel size (handles remainder)
    for (i=0; i<m; i+=mc) {       // Loop over mc-sized row blocks of A/C
        ib = min(m-i, mc);
        InnerKernel(...);         // Compute ib×n block of C
    }
}
```

This creates **L2 cache blocking** where `pb×ib` elements of A fit in L2 cache.

Data Packing Transformation The key innovation is converting matrices to **packed formats**:

`PackMatrixA` (called when `j==0` in inner loop): - Packs 4 rows of A's current panel into contiguous memory - Original access pattern: `A(0,j), A(1,j), A(2,j), A(3,j)` for each column j - Packed format: `[a00 a10 a20 a30 | a01 a11 a21 a31 | ...]` (4×k block in column-major) - **Enables unit-stride SIMD loads** via `_mm_load_pd()` for two consecutive rows

`PackMatrixB` (called once per column block when `first_time`): - Packs 4 columns of B's current panel - Uses pointer arithmetic: `b_i0_pntr++` advances down a column (unit stride) - Packed format: `[b00 b01 b02 b03 | b10 b11 b12 b13 | ...]` (k×4 block in row-major) - **Enables efficient broadcast** via `_mm_loaddup_pd()` (SSE3)

Vectorized Micro-Kernel (`AddDot4x4`)

Vector Register Allocation

```
v2df_t c_00_c_10_vreg, c_01_c_11_vreg, ...;  // Accumulator registers
```

Each `v2df_t` holds **two double-precision values** in a 128-bit SSE register. The naming convention `c_00_c_10` indicates it accumulates: - `c_00_c_10_vreg.d[0]` → `C(0,0)` - `c_00_c_10_vreg.d[1]` → `C(1,0)`

This **register blocking** computes 4×4 output elements using: - 8 accumulator registers (2 per output row pair) - 2 packed A registers (`a_0p_a_1p_vreg`, `a_2p_a_3p_vreg`) - 4 broadcast B registers (`b_p0_vreg` to `b_p3_vreg`)

SSE Intrinsics Breakdown

```
a_0p_a_1p_vreg.v = _mm_load_pd((double *) a);  // Load 2 doubles (rows 0,1)
a_2p_a_3p_vreg.v = _mm_load_pd((double *) (a+2)); // Load rows 2,3
b_p0_vreg.v = _mm_loaddup_pd((double *) b);    // Broadcast b[p,0]
```

- `_mm_load_pd()`: Requires 16-byte alignment (packed A ensures this)
- `_mm_loaddup_pd()`: Duplicates a scalar to both halves of register (SSE3)
- The multiplication `a_0p_a_1p_vreg.v * b_p0_vreg.v` is **SIMD FMADD** (fused multiply-add)

Computational Pattern For each iteration `p`: 1. Load 4 elements from packed A (two SIMD loads) 2. Broadcast 4 elements from packed B (four load-and-duplicate) 3. Perform 8 SIMD multiply-adds: `c00_c10 += [a0p, a1p] * b_p0  // First two rows × column 0    c01_c11 += [a0p, a1p] * b_p1  // First two rows × column 1    ...    c20_c30 += [a2p, a3p] * b_p0  // Next two rows × column 0`

Memory Access Patterns

- **Packed arrays**: Enable unit-stride access during computation
- **B packing**: Converts column-major to row-major for the 4-column block
- **A packing**: Keeps column-major but removes large `lda` stride
- **Static buffer for B**: `static double packedB[kc*nb]` persists across calls (cache-friendly but not thread-safe)

3. HPC Context

Cache Optimization Principles

1. **Temporal Locality**: The 4×4 micro-kernel keeps 8 accumulator registers in CPU registers (fastest memory)
2. **Spatial Locality**: Packed arrays ensure cache lines are fully utilized
3. **Cache Blocking**:
   - `mc×kc` block of A should fit in L2 cache
   - `kc×4` block of B should fit in L1 cache
   - 4×4 block of C fits in registers

Arithmetic Intensity Analysis For a 4×4×k micro-kernel: - **Operations**: `2 * 4 * 4 * k = 32k` flops (FMADD counts as 2) - **Data movement**: - A: `4 * k * 8 bytes = 32k bytes` - B: `4 * k * 8 bytes = 32k bytes` - C: `4 * 4 * 8 bytes = 128 bytes` (loaded/stored once) - **Approximate arithmetic intensity**: `~32k/(64k + 128)` 0.5 flops/byte - **Potential bottleneck**: Memory bandwidth unless k is large enough

SIMD Efficiency

- **Vectorization width**: 2 (double-precision in SSE)
- **Utilization**: 100% - all SIMD lanes used
- **Instruction mix**:
  - 2 loads + 4 load-duplicate per iteration
  - 8 FMADD operations per iteration
  - Potential for **4:1 compute

Source Code Implementation

```
/* Create macros so that the matrices are stored in column-major order */

#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Block sizes */
#define mc 256
#define kc 128
#define nb 1000

#define min( i, j ) ( (i)<(j) ? (i): (j) )

/* Routine for computing C = A * B + C */

void AddDot4x4( int, double *, int, double *, int, double *, int );
void PackMatrixA( int, double *, int, double * );
void PackMatrixB( int, double *, int, double * );
void InnerKernel( int, int, int, double *, int, double *, int, double *, int, int );

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, p, pb, ib;

  /* This time, we compute a mc x n block of C by a call to the InnerKernel */

  for ( p=0; p<k; p+=kc ){
    pb = min( k-p, kc );
    for ( i=0; i<m; i+=mc ){
      ib = min( m-i, mc );
      InnerKernel( ib, n, pb, &A( i,p ), lda, &B(p, 0 ), ldb, &C( i,0 ), ldc, i==0 );
    }
  }
}

void InnerKernel( int m, int n, int k, double *a, int lda,
                                       double *b, int ldb,
                                       double *c, int ldc, int first_time )
{
  int i, j;
  double
    packedA[ m * k ];
```

```
  static double
    packedB[ kc*nb ];    /* Note: using a static buffer is not thread safe... */

  for ( j=0; j<n; j+=4 ){        /* Loop over the columns of C, unrolled by 4 */
    if ( first_time )
      PackMatrixB( k, &B( 0, j ), ldb, &packedB[ j*k ] );
    for ( i=0; i<m; i+=4 ){        /* Loop over the rows of C */
      /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
     one routine (four inner products) */
      if ( j == 0 )
    PackMatrixA( k, &A( i, 0 ), lda, &packedA[ i*k ] );
      AddDot4x4( k, &packedA[ i*k ], 4, &packedB[ j*k ], k, &C( i,j ), ldc );
    }
  }
}

void PackMatrixA( int k, double *a, int lda, double *a_to )
{
  int j;

  for( j=0; j<k; j++){  /* loop over columns of A */
    double
      *a_ij_pntr = &A( 0, j );

    *a_to     = *a_ij_pntr;
    *(a_to+1) = *(a_ij_pntr+1);
    *(a_to+2) = *(a_ij_pntr+2);
    *(a_to+3) = *(a_ij_pntr+3);

    a_to += 4;
  }
}

void PackMatrixB( int k, double *b, int ldb, double *b_to )
{
  int i;
  double
    *b_i0_pntr = &B( 0, 0 ), *b_i1_pntr = &B( 0, 1 ),
    *b_i2_pntr = &B( 0, 2 ), *b_i3_pntr = &B( 0, 3 );

  for( i=0; i<k; i++){  /* loop over rows of B */
    *b_to++ = *b_i0_pntr++;
    *b_to++ = *b_i1_pntr++;
    *b_to++ = *b_i2_pntr++;
    *b_to++ = *b_i3_pntr++;
  }
```

```
}

#include <mmintrin.h>
#include <xmmintrin.h>  // SSE
#include <pmmintrin.h>  // SSE2
#include <emmintrin.h>  // SSE3

typedef union
{
  __m128d v;
  double d[2];
} v2df_t;

void AddDot4x4( int k, double *a, int lda,  double *b, int ldb, double *c, int ldc )
{
  /* So, this routine computes a 4x4 block of matrix A

           C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ).
           C( 1, 0 ), C( 1, 1 ), C( 1, 2 ), C( 1, 3 ).
           C( 2, 0 ), C( 2, 1 ), C( 2, 2 ), C( 2, 3 ).
           C( 3, 0 ), C( 3, 1 ), C( 3, 2 ), C( 3, 3 ).

     Notice that this routine is called with c = C( i, j ) in the
     previous routine, so these are actually the elements

           C( i  , j ), C( i  , j+1 ), C( i  , j+2 ), C( i  , j+3 )
           C( i+1, j ), C( i+1, j+1 ), C( i+1, j+2 ), C( i+1, j+3 )
           C( i+2, j ), C( i+2, j+1 ), C( i+2, j+2 ), C( i+2, j+3 )
           C( i+3, j ), C( i+3, j+1 ), C( i+3, j+2 ), C( i+3, j+3 )

     in the original matrix C

     And now we use vector registers and instructions */

  int p;
  v2df_t
    c_00_c_10_vreg,    c_01_c_11_vreg,    c_02_c_12_vreg,    c_03_c_13_vreg,
    c_20_c_30_vreg,    c_21_c_31_vreg,    c_22_c_32_vreg,    c_23_c_33_vreg,
    a_0p_a_1p_vreg,
    a_2p_a_3p_vreg,
    b_p0_vreg, b_p1_vreg, b_p2_vreg, b_p3_vreg;

  c_00_c_10_vreg.v = _mm_setzero_pd();
  c_01_c_11_vreg.v = _mm_setzero_pd();
  c_02_c_12_vreg.v = _mm_setzero_pd();
  c_03_c_13_vreg.v = _mm_setzero_pd();
```

```
  c_20_c_30_vreg.v = _mm_setzero_pd();
  c_21_c_31_vreg.v = _mm_setzero_pd();
  c_22_c_32_vreg.v = _mm_setzero_pd();
  c_23_c_33_vreg.v = _mm_setzero_pd();

  for ( p=0; p<k; p++ ){
    a_0p_a_1p_vreg.v = _mm_load_pd( (double *) a );
    a_2p_a_3p_vreg.v = _mm_load_pd( (double *) ( a+2 ) );
    a += 4;

    b_p0_vreg.v = _mm_loaddup_pd( (double *) b );       /* load and duplicate */
    b_p1_vreg.v = _mm_loaddup_pd( (double *) (b+1) );   /* load and duplicate */
    b_p2_vreg.v = _mm_loaddup_pd( (double *) (b+2) );   /* load and duplicate */
    b_p3_vreg.v = _mm_loaddup_pd( (double *) (b+3) );   /* load and duplicate */

    b += 4;

    /* First row and second rows */
    c_00_c_10_vreg.v += a_0p_a_1p_vreg.v * b_p0_vreg.v;
    c_01_c_11_vreg.v += a_0p_a_1p_vreg.v * b_p1_vreg.v;
    c_02_c_12_vreg.v += a_0p_a_1p_vreg.v * b_p2_vreg.v;
    c_03_c_13_vreg.v += a_0p_a_1p_vreg.v * b_p3_vreg.v;

    /* Third and fourth rows */
    c_20_c_30_vreg.v += a_2p_a_3p_vreg.v * b_p0_vreg.v;
    c_21_c_31_vreg.v += a_2p_a_3p_vreg.v * b_p1_vreg.v;
    c_22_c_32_vreg.v += a_2p_a_3p_vreg.v * b_p2_vreg.v;
    c_23_c_33_vreg.v += a_2p_a_3p_vreg.v * b_p3_vreg.v;
  }

  C( 0, 0 ) += c_00_c_10_vreg.d[0];  C( 0, 1 ) += c_01_c_11_vreg.d[0];
  C( 0, 2 ) += c_02_c_12_vreg.d[0];  C( 0, 3 ) += c_03_c_13_vreg.d[0];

  C( 1, 0 ) += c_00_c_10_vreg.d[1];  C( 1, 1 ) += c_01_c_11_vreg.d[1];
  C( 1, 2 ) += c_02_c_12_vreg.d[1];  C( 1, 3 ) += c_03_c_13_vreg.d[1];

  C( 2, 0 ) += c_20_c_30_vreg.d[0];  C( 2, 1 ) += c_21_c_31_vreg.d[0];
  C( 2, 2 ) += c_22_c_32_vreg.d[0];  C( 2, 3 ) += c_23_c_33_vreg.d[0];

  C( 3, 0 ) += c_20_c_30_vreg.d[1];  C( 3, 1 ) += c_21_c_31_vreg.d[1];
  C( 3, 2 ) += c_22_c_32_vreg.d[1];  C( 3, 3 ) += c_23_c_33_vreg.d[1];
}
```
