PolyBench 4.0

Tomofumi Yuki
Inria / LIP / ENS Lyon
tomofumi.yuki@inria.fr

Louis-Noël Pouchet
Ohio State University
pouchet@cse.ohio-state.edu

February 18, 2015

Contents

1 PolyBench 4.0 Kernels

This document describes computations performed in PolyBench kernels from mathematical stand point. The description is detached from implementation decisions to provide a single and consistent point of reference for implementations in various languages.

1.1 Data Mining

`covariance`

Computes the covariance, a measure from statistics that show how linearly related two variables are.

It takes the following as input,

- `data`: $N \times M$ matrix that represents N data points, each with M attributes,

and gives the following as output:

- `cov`: $M \times M$ matrix where the i, j-th element is the covariance between i and j. The matrix is symmetric.

Covariance is defined to be the mean of the product of deviations for i and j:

$$\texttt{cov}(i,j) = \frac{\sum_{k=0}^{N-1} (\texttt{data}(k,i) - \texttt{mean}(i))(\texttt{data}(k,j) - \texttt{mean}(j))}{N-1}$$

where

$$\texttt{mean}(x) = \frac{\sum_{k=0}^{N-1} \texttt{data}(k,x)}{N}$$

Note that the above computes *sample covariance* where the denominator is $N - 1$.

`correlation`

Correlation computes the correlation coefficients (Pearson's), which is normalized covariance.

It takes the following as input,

- `data`: $N \times M$ matrix that represents N data points, each with M attributes,

and gives the following as output:

- `corr`: $M \times M$ matrix where the i, j-th element is the correlation coefficient between i and j. The matrix is symmetric.

Correlation is defined as the following,

$$\texttt{corr}(i,j) = \frac{\texttt{cov}(i,j)}{\texttt{stddev}(i)\texttt{stddev}(j)}$$

where

$$\texttt{stddev}(x) = \sqrt{\frac{\sum_{k=0}^{N-1} (\texttt{data}(k,x) - \texttt{mean}(x))^2}{N}}$$

`cov` and `mean` are defined in `covariance`[1].

[1] However, the denominator when computing covariance is N for correlation.

1.2 BLAS Routines

`gemm`

Generalized Matrix Multiply from BLAS [4].

It takes the following as inputs,

- α, β: scalars
- `A`: $P \times Q$ matrix
- `B`: $Q \times R$ matrix
- `C`: $P \times R$ matrix

and gives the following as output:

- `Cout`: $P \times R$ array, where `Cout` $= \alpha AB + \beta C$

Note that the output `Cout` is to be stored in place of the input array `C`. The BLAS parameters used are `TRANSA = TRANSB = 'N'`, meaning both `A` and `B` are not transposed.

`gemver`

Multiple matrix-vector multiplication from updated BLAS [1]. However it is not part of the current BLAS distribution.

It takes the following as inputs,

- α, β: scalars
- `A`: $N \times N$ matrix
- `u1`, `u2`, `v1`, `v2`, `y`, `z`: vectors of length N

and gives the following as outputs:

- `A'`: $N \times N$ matrix, where $A' = A + u1.v1 + u2.v2$
- `x`: vector of length N, where $x = \beta A'y + z$
- `w`: vector of length N, where $w = \alpha A'x$

`gesummv`

Summed matrix-vector multiplications from updated BLAS [1]. However it is not part of the current BLAS distribution.

It takes the following as inputs,

- α, β: scalars
- `A, B`: $N \times N$ matrix
- `x`: vector of length N

and gives the following as outputs:

- `y`: vector of length N, where $y = \alpha Ax + \beta Bx$

`symm`

Symmetric matrix matrix multiplication from BLAS [4].
It takes the following as inputs,

- α, β: scalars
- `A`: $M \times M$ symmetric matrix
- `B,C`: $M \times N$ matrices

and gives the following as output:

- `Cout`: $M \times N$ matrix, where `Cout` $= \alpha AB + \beta C$

Note that the output `Cout` is to be stored in place of the input array `C`. The matrix A is stored as a triangular matrix in BLAS. The configuration used are `SIDE = 'L'` and `UPLO = 'L'`, meaning the multiplication is from the left, and the symmetric matrix is stored as a lower triangular matrix.

`syrk`

Symmetric rank k update from updated BLAS [1].
It takes the following as inputs,

- α, β: scalars
- `A`: $N \times M$ matrix
- `C`: $N \times N$ symmetric matrix

and gives the following as output:

- `Cout`: $N \times N$ matrix, where `Cout` $= \alpha AA^T + \beta C$

Note that the output `Cout` is to be stored in place of the input array `C`. The matrix C is stored as a triangular matrix in BLAS, and the result is also triangular. The configuration used are `TRANS = 'N'` and `UPLO = 'L'` meaning the `A` matrix is not transposed, and `C` matrix stores the symmetric matrix as lower triangular matrix.

`syr2k`

Symmetric rank 2k update from updated BLAS [1].
It takes the following as inputs,

- α, β: scalars
- `A,B`: $N \times M$ matrices
- `C`: $N \times N$ symmetric matrix

and gives the following as output:

- `Cout`: $N \times N$ matrix, where `Cout` $= \alpha AB^T + \alpha BA^T + \beta C$

Note that the output `Cout` is to be stored in place of the input array `C`. The matrix C is stored as a triangular matrix in BLAS, and the result is also triangular. The configuration used are `TRANS = 'N'` and `UPLO = 'L'` meaning the `A` matrix is not transposed, and `C` matrix stores the symmetric matrix as lower triangular matrix.

`trmm`

Triangular matrix multiplication.

It takes the following as inputs,

- `A`: $N \times N$ lower triangular matrix
- `B`: $N \times N$ matrix

and gives the following as output:

- `Bout`: $N \times N$ matrix, where `Bout` $= AB$

Note that the output `Bout` is to be stored in place of the input array `B`. The configurations used are `SIDE = 'L'`, `UPLO = 'L'`, `TRANSA = 'T'`, and `DIAG = 'U'`, meaning the multiplication is from the left, the matrix is lower triangular, untransposed with unit diagonal.

1.3 Linear Algebra Kernels

`2mm`

Linear algebra kernel that consists of two matrix multiplications.

It takes the following as inputs,

- α, β: scalars
- `A`: $P \times Q$ matrix
- `B`: $Q \times R$ matrix
- `C`: $R \times S$ matrix
- `D`: $P \times S$ matrix

and gives the following as output:

- `E`: $P \times S$ matrix, where $E = \alpha ABC + \beta D$

`3mm`

Linear algebra kernel that consists of three matrix multiplications.

It takes the following as inputs,

- `A`: $P \times Q$ matrix
- `B`: $Q \times R$ matrix
- `C`: $R \times S$ matrix
- `D`: $S \times T$ matrix

and gives the following as output:

- `G`: $P \times T$ matrix, where $G = (A.B).(C.F)$

`atax`

Computes A^T times Ax.

It takes the following as inputs,

- `A`: $M \times N$ matrix
- `x`: vector of length N

and gives the following as output:

- `y`: vector of length N, where $y = A^T(Ax)$

`bicg`

Kernel of BiCGSTAB (BiConjugate Gradient STABilized method).

It takes the following as inputs,

- `A`: $N \times M$ matrix
- `p`: vector of length M
- `r`: vector of length N

and gives the following as output:

- `q`: vector of length N, where $q = Ap$
- `s`: vector of length M, where $s = A^T r$

`doitgen`

Kernel of Multiresolution ADaptive NumErical Scientific Simulation (MADNESS)[2]. The kernel is taken from the modified version used by You et al. [7].

It takes the following as inputs,

- `A`: $R \times Q \times S$ array
- `x`: $P \times S$ array

and gives the following as output:

- `Aout`: $R \times Q \times P$ array

$$\texttt{A}(r, q, p) = \sum_{s=0}^{S-1} \texttt{A}(r, q, s)\texttt{x}(p, s)$$

Note that the output `Aout` is to be stored in place of the input array `A` in the original code. Although it is not mentioned anywhere, the computation does not make sense if $P \neq S$.

`mvt`

Matrix vector multiplication composed with another matrix vector multiplication but with transposed matrix.

It takes the following as inputs,

- `A`: $N \times N$ matrix
- `y1, y2`: vectors of length N

and gives the following as outputs:

- `x1`: vector of length N, where $x1 = x1 + Ay1$
- `x2`: vector of length N, where $x2 = x2 + A^T y2$

[2] `http://www.csm.ornl.gov/ccsg/html/projects/madness.html`

1.4 Linear Algebra Solvers

`cholesky`

Cholesky decomposition, which decomposes a matrix to triangular matrices. Only applicable when the input matrix is positive-definite.

It takes the following as input,

- `A`: $N \times N$ positive-definite matrix

and gives the following as output:

- `L`: $N \times N$ lower triangular matrix such that $A = LL^T$

The `C` reference implementation uses CholeskyBanachiewicz algorithm. The algorithm computes the following, where the computation starts from the upper-left corner of L and proceeds row by row.

$$\mathtt{L}(i,j) = \begin{cases} i = j & : \sqrt{\mathtt{A}(i,i) - \sum_{k=0}^{i-1} \mathtt{L}(i,k)^2} \\ i > j & : \dfrac{\mathtt{A}(i,j) - \sum_{k=0}^{j-1} \mathtt{L}(i,k)\mathtt{L}(k,j)}{\mathtt{L}(j,j)} \end{cases}$$

`durbin`

Durbing is an algorithm for solving Yule-Walker equations, which is a special case of Toelitz systems.

It takes the following as input,

- `r`: vector of length N.

and gives the following as output:

- `y`: vector of length N

such that $Ty = -r$ where T is a symmetric, unit-diagonal, Toeplitz matrix defined by the vector $[1, r_0, ..., r_{N-1}]$.

The C reference implementation is a direct implementation of the algorithm described in a book by Golub and Van Loan [5]. The book mentions that a vector can be removed to use less space, but the implementation retains this vector.

`gramschmidt`

QR Decomposition with Modified Gram Schmidt.

It takes the following as input,

- `A`: $M \times N$ rank N matrix ($M > N$).

and gives the following as outputs:

- `Q`: $M \times N$ orthogonal matrix
- `R`: $N \times N$ upper tiangular matrix

such that $A = QR$.

The algorithm is described in a techreport by Walter Gander: `http://www.inf.ethz.ch/personal/gander/`.

`lu`

LU decomposition without pivoting.

It takes the following as input,

- `A`: $N \times N$ matrix

and gives the following as outputs:

- `L`: $N \times N$ lower triangular matrix
- `U`: $N \times N$ upper triangular matrix

such that $A = LU$.

`L` and `U` are computed as follows:

$$\mathtt{U}(i,j) = \mathtt{A}(i,j) - \sum_{k=0}^{i-1} \mathtt{L}(i,k)\mathtt{U}(k,j)$$

$$\mathtt{L}(i,j) = \frac{\mathtt{A}(i,j) - \sum_{k=0}^{j-1} \mathtt{L}(i,k)\mathtt{U}(k,j)}{\mathtt{U}(j,j)}$$

`ludcmp`

This kernel solves a system of linear equations using LU decomposition followed by forward and backward substitutions.

It takes the following as inputs,

- `A`: $N \times N$ matrix
- `b`: vector of length N

and gives the following as output:

- `x`: vector of length N, where $Ax = b$

The matrix A is first decomposed into L and U using the same algorithm as in `lu`. Then the two triangular systems are solved to find x as follows:

$$Ax = b \Rightarrow LUx = b \Rightarrow \begin{cases} Ly = b \\ Ux = y \end{cases}$$

The forward and backward substitutions are as follows:

$$\mathtt{y}(i) = \frac{\mathtt{b}(i) - \sum_{j=0}^{i-1} \mathtt{L}(i,j)\mathtt{y}(j)}{\mathtt{L}(i,i)}$$

$$\mathtt{x}(i) = \frac{\mathtt{y}(i) - \sum_{j=0}^{i-1} \mathtt{U}(i,j)\mathtt{x}(j)}{\mathtt{U}(i,i)}$$

`trisolv`

Triangular matrix solver using forward substitution.

It takes the following as inputs,

- `L`: $N \times N$ lower triangular matrix
- `b`: vector of length N

and gives the following as output:

- `x`: vector of length N, where $Lx = b$

The forward substitution is as follows:

$$\mathtt{x}(i) = \frac{\mathtt{b}(i) - \sum_{j=0}^{i-1} \mathtt{L}(i,j)\mathtt{x}(j)}{\mathtt{L}(i,i)}$$

1.5 Medley

`deriche`

`deriche` implements the Deriche recursive filter, which is a generic filter that can be used for both edge detection and smoothing [2, 3]. The 2D version take advantage of the separability and is implemented as horizontal passes followed by vertical passes over an image.

It takes the following as input,

- `x`: $W \times H$ image.
- α: A parameter of the filter that is related to the size of the convolution.
- $a_1, \ldots, a_8, b_1, b_2, c_1, c_2$: Coefficients of the filter that determine the behavior of the filter.

and gives the following as output:

- `y`: The processed image. How the image is processed depends on the coefficients used.

The generic 2D implementation reproduced from the original article [3] is as follows:

Horizontal Pass

$$y_1(i,j) = a_1\mathtt{x}(i,j) + a_2\mathtt{x}(i,j-1) + b_1 y_1(i,j-1) + b_2 y_1(i,j-2)$$
$$y_2(i,j) = a_3\mathtt{x}(i,j+1) + a_4\mathtt{x}(i,j+2) + b_1 y_2(i,j+1) + b_2 y_2(i,j+2)$$
$$r(i,j) = c_1\left(y_1(i,j) + y_2(i,j)\right)$$

Vertical Pass

$$z_1(i,j) = a_5 r(i,j) + a_6 r(i-1,j) + b_1 y_1(i-1,j) + b_2 y_1(i-2,j)$$
$$z_2(i,j) = a_7 r(i+1,j) + a_8 r(i+2,j) + b_1 y_1(i+1,j) + b_2 y_1(i+2,j)$$
$$\mathtt{y}(i,j) = c_2\left(z_1(i,j) + z_2(i,j)\right)$$

The C implementation in PolyBench use the following parameters for the filter:

- $\alpha = 0.25$
- $a_1 = a_5 = k$
- $a_2 = a_6 = ke^{-\alpha}(\alpha - 1)$

- $a_3 = a_7 = ke^{-\alpha}(\alpha + 1)$
- $a_4 = a_8 = -ke^{-2\alpha}$
- $b_1 = 2e^{-\alpha}$
- $b_2 = -e^{-2\alpha}$
- $c_1 = c_2 = 1$

where k is computed as

$$k = \frac{(1 - e^{-\alpha})^2}{1 + 2\alpha e^{-\alpha} - e^{-2\alpha}}$$

which implements the smoothing filter.

`floyd-warshall`

Floyd-Warshall computes the shortest paths between each pair of nodes in a graph. The following only computes the shortest path lengths, which is a typical use of this method.

It takes the following as input,

- `w`: $N \times N$ matrix, where the i, jth entry represents the cost of taking an edge from i to j. Set to infinity if there is no edge connecting i to j.

and gives the following as output:

- `paths`: $N \times N$ matrix, where the i, jth entry represents the shortest path length from i to j.

The shortest path lengths are computed recusrively as follows:

$$\mathtt{p}(k, i, j) = \begin{cases} k = -1 & : \mathtt{w}(i, j) \\ 0 \leq k < N & : min(\mathtt{p}(k-1, i, j), \mathtt{p}(k-1, i, k) + \mathtt{p}(k-1, k, j) \end{cases}$$

where the final output $\mathtt{paths}(i, j) = \mathtt{p}(N - 1, i, j)$.

`nussinov`

Nussinov is an algorithm for predicting RNA folding, and is an instance of dynamic programming.

It takes the following as input,

- `seq`: RNA sequence of length N. The valid entries are one of 'A' 'G' 'C' 'T'. (or 'U' in place of 'T').

and gives the following as output:

- `table`: $N \times N$ triangular matrix, which is the dynamic programming table.

The table is filled using the following formula:

$$\mathtt{table}(i, j) = max \begin{pmatrix} \mathtt{table}(i+1, j) \\ \mathtt{table}(i, j-1) \\ \mathtt{table}(i+1, j-1) + w(i, j) \\ max_{i<k<j} \left(\mathtt{table}(i, k) + \mathtt{table}(k+1, j)\right) \end{pmatrix}$$

where w is the scoring function that evaluate the pair of sequences $\mathtt{seq}[i]$ and $\mathtt{seq}[j]$. For Nussinov algorithm, the scoring function returns 1 if the sequences are complementary (either 'A' with 'T' or 'G' with 'C'), and 0 otherwise.

1.6 Stencils

Stencil computations iteratively update a grid of data using the same pattern of computation. For stencil computations, we denote the time step of the iterative process as super scripts to simplify the description.

In the reference implementations, boundary conditions are simplified by not updating them. The updates are only performed when all of its operands are available (i.e., no out-of-bounds accesses).

`adi`

Alternating Direction Implicit method for 2D heat diffusion. The main strength of ADI over other methods is that it decomposes a 2D problem to two 1D problems, allowing direct solutions to the sub-problems with lower cost. For 2D heat equations, each sub-problem becomes a tridiagonal system of equations.

From a "stencil" point of view, this can be seen as taking the heat equation of the form:

$$\mathtt{u}^{t+1} = f\left(\begin{array}{ccc} & \mathtt{u}^t_{i,j+1}, & \\ \mathtt{u}^t_{i+1,j}, & \mathtt{u}^t_{i,j}, & \mathtt{u}^t_{i-1,j} \\ & \mathtt{u}^t_{i,j-1} & \end{array}\right)$$

and splitting it two two steps:

$$\mathtt{u}^{t+\frac{1}{2}} = g\left(\mathtt{u}^t_{i+1,j}, \mathtt{u}^t_{i,j}, \mathtt{u}^t_{i-1,j}\right)$$
$$\mathtt{u}^{t+1} = h\left(\mathtt{u}^{t+\frac{1}{2}}_{i,j+1}, \mathtt{u}^{t+\frac{1}{2}}_{i,j}, \mathtt{u}^{t+\frac{1}{2}}_{i,j-1}\right)$$

The C reference implementation is based on a Fortlan code in a paper by Li and Kedem [6]. In this implementation, the coefficients are also computed since the parameters configure the resolution and not the size of the space/time domain.

`fdtd-2d`

Simplified Finite-Difference Time-Domain method for 2D data, which models electric and mangetic fields based on Maxwell's equations. In particular, the polarization used here is TE^z; Transverse Electric in z direction. It is a stencil involving three variables, `Ex`, `Ey`, and `Hz`. `Ex` and `Ey` are electric fields varying in x and y axes, where `Hz` is the magnetic field along z axis. Fields along other axes are either zero or static, and are not modeled.

$$\mathtt{Hz}^t_{(i,j)} = C_{hzh}\mathtt{Hz}^{t-1}_{(i,j)} + C_{hze}(\mathtt{Ex}^{t-1}_{(i,j+1)} - \mathtt{Ex}^{t-1}_{(i,j)} - \mathtt{Ey}^{t-1}_{(i+1,j)} - \mathtt{Ey}^{t-1}_{(i,j)})$$
$$\mathtt{Ex}^t_{(i,j)} = C_{exe}\mathtt{Ex}^{t-1}_{(i,j)} + C_{exh}(\mathtt{Hz}^t_{(i,j)} - \mathtt{Hz}^t_{(i,j-1)})$$
$$\mathtt{Ey}^t_{(i,j)} = C_{eye}\mathtt{Ey}^{t-1}_{(i,j)} + C_{eyh}(\mathtt{Hz}^t_{(i,j)} - \mathtt{Hz}^t_{(i-1,j)})$$

Variables C_{xxx} are coefficients that may be different depending on the location within the descritized space. In PolyBench, it is simplified as scalar coefficients.

`heat-3d`

Heat Equation over 3D space. The main update is as follows:

$$\mathtt{data}^t_{(i,j,k)} = \mathtt{data}^{t-1}_{(i,j,k)} + \left(\begin{array}{l} 0.125\left(\mathtt{data}^{t-1}_{(i+1,j,k)} - 2\mathtt{data}^{t-1}_{(i,j,k)} + \mathtt{data}^{t-1}_{(i-1,j,k)}\right) + \\ 0.125\left(\mathtt{data}^{t-1}_{(i,j+1,k)} - 2\mathtt{data}^{t-1}_{(i,j,k)} + \mathtt{data}^{t-1}_{(i,j-1,k)}\right) + \\ 0.125\left(\mathtt{data}^{t-1}_{(i,j,k+1)} - 2\mathtt{data}^{t-1}_{(i,j,k)} + \mathtt{data}^{t-1}_{(i,j,k-1)}\right) \end{array}\right)$$

The C reference implementation is originally from Pochoir distribution.

`jacobi-1d`

Jacobi style stencil computation over 1D data with 3-point stencil pattern. It originally comes from the Jacobi method for solving system of equations. However, Jacobi-style stencils may refer to computations with some stencil pattern that use values computed in the previous time step, which is a characteristic of Jacobi method. The computation in PolyBench is simplified as simply taking the average of three points.

$$\texttt{data}^{t}_{(i)} = \texttt{mean}\left(\texttt{data}^{t-1}_{(i)} + \texttt{data}^{t-1}_{(i-1)} + \texttt{data}^{t-1}_{(i+1)}\right)$$

`jacobi-2d`

Jacobi-style stencil computation over 2D data with 5-point stencil pattern. The computation is simplified as simply taking the average of five points.

$$\texttt{data}^{t}_{(i,j)} = \texttt{mean}\left(\texttt{data}^{t-1}_{(i,j)} + \texttt{data}^{t-1}_{(i-1,j)} + \texttt{data}^{t-1}_{(i+1,j)} + \texttt{data}^{t-1}_{(i,j-1)} + \texttt{data}^{t-1}_{(i,j+1)}\right)$$

`seidel-2d`

Gauss-Seidel style stencil computation over 2D data with 9-point stencil pattern. Similar to Jacobi style stencils, Gauss-Seidel style stencil comes from Gauss-Seidel method for solving systems of linear equations. The main difference is that values computed at the same time step is used, which changes its convergence and other properties. The computation in PolyBench is simplified as simply taking the average of nine points.

$$\texttt{data}^{t}_{(i,j)} = \texttt{mean}\left(\begin{array}{c} \texttt{data}^{t-1}_{(i-1,j-1)}, \texttt{data}^{t-1}_{(i-1,j)}, \texttt{data}^{t-1}_{(i-1,j+1)}, \\ \texttt{data}^{t-1}_{(i,j-1)}, \texttt{data}^{t-1}_{(i,j)}, \texttt{data}^{t-1}_{(i-1,j+1)}, \\ \texttt{data}^{t-1}_{(i+1,j-1)}, \texttt{data}^{t-1}_{(i+1,j)}, \texttt{data}^{t-1}_{(i+1,j+1)} \end{array}\right)$$

2 C Implementation

The C implementation follows the algorithms described in this document. There are many different ways to implement each of them, and some guidelines used when selecting the implementation details are given below.

- The general principle is to make the kernels as simple as possible for research compilers to handle. In 3.3, one exception is made and reverse loops are now included in some of the kernels. The main motivation is that reverting these loops makes the kernels much more unnatural and complicated to understand. The dataflow analysis implemented in Integer Set Library allow reverse loops to be handled very easily.
- For BLAS kernels, it is kept as close to the original Fortran implementation as possible. However, they are not identical in most cases due to the difference in array layout. Some BLAS kernels use scalars preventing loops to be parallelized unless the scalars are privatized.
- The memory allocations used in the implementation is intended to be *a* natural allocation for each kernel. As a rule of thumb, the arrays are made as compact as possible, while keeping them rectangular. Most dense linear algebra kernels performs in-place computation when applicable.

Known Issues

- `correlation` output is all 1 due to the property of correlation combined with the input generation.

Pre-defined Problem Sizes

PolyBench/C 4.0 comes with 5 pre-defined problem sizes. The problem sizes are primarily derived from the memory requirements such that different levels of the memory hierarchy are exercised.

- `MINI`: Less than 16KB of memory. The problem may fit within the L1 (last level) cache.
- `SMALL`: Around 128KB of memory. The problem should not fit within the L1 cache, but may fit L2.
- `MEDIUM`: Around 1MB of memory. The problem should not fit within the L2 cache, but may fit L3.
- `LARGE`: Around 25MB of memory. The problem should not fit within the L3 cache.
- `EXTRALARGE`: Around 120MB of memory.

Additional remarks:

- The default problem size is `LARGE`.
- It is recommended to revise your problem sizes when the data types are changed.
- Different parameters are given different values by default.
- Sizes for `deriche` are taken from commonly used image resolutions.
- Two kernels (`durbin` and `jacobi-1d`) have $O(n)$ memory requirement, and thus the above rules do not apply. The problem sizes for these kernels match those of other solvers (e.g., `cholesky`).

Summary of Characteristics

Table 1 summarizes the number of operations and memory usage of each kernel. The problem size parameters are assumed to take the same value n.

References

[1] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[2] R. Deriche. Using canny's criteria to derive a recursively implemented optimal edge detector. *International Journal of Computer Vision*, 1(2):167–187, 1987.

[3] R. Deriche. Fast algorithms for low-level vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(1):78–87, Jan 1990.

[4] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[5] G. H. Golub and C. F. Van Loan. *Matrix Computations*, volume 3. Johns Hopkins University Press, 2012.

[6] P. Lee and Z. M. Kedem. Automatic data and computation decomposition on distributed memory parallel computers. *ACM Transactions on Programming Languages and Systems*, 24(1):1–50, Jan. 2002.

[7] H. You, K. Seymour, J. Dongarra, and S. Moore. Empirical tuning of a multiresolution analysis kernel using a specialized code generator. Technical report, ICL-UT-07-02, March 2007.

Table 1: Table

kernel	Operations	Memory	O(Ops)	O(Mem)
`correlation`	$n^3 + 8n^2 + 3n$	$2n^2 + 2n$	$O(n^3)$	$O(n^2)$
`covariance`	$n^3 + 4n^2 + 2n$	$2n^2 + n$	$O(n^3)$	$O(n^2)$
`2mm`	$5n^3 + n^2$	$4n^2$	$O(n^3)$	$O(n^2)$
`3mm`	$6n^3$	$4n^2$	$O(n^3)$	$O(n^2)$
`atax`	$4n^2$	$n^2 + 3n$	$O(n^2)$	$O(n^2)$
`bicg`	$4n^2$	$n^2 + 4n$	$O(n^2)$	$O(n^2)$
`doitgen`	$2n^4$	$n^3 + n^2 + n$	$O(n^4)$	$O(n^3)$
`mvt`	$4n^2$	$n^2 + 4n$	$O(n^2)$	$O(n^2)$
`gemm`	$3n^3 + n^2$	$3n^2$	$O(n^3)$	$O(n^2)$
`gemver`	$10n^2 + n$	$n^2 + 8n$	$O(n^2)$	$O(n^2)$
`gesummv`	$4n^2 + 3n$	$2n^2 + 3n$	$O(n^2)$	$O(n^2)$
`symm`	$\frac{5}{2}n^3 + \frac{5}{2}n^2$	$3n^2$	$O(n^3)$	$O(n^2)$
`syr2k`	$6n^3 + n^2$	$3n^2$	$O(n^3)$	$O(n^2)$
`syrk`	$\frac{3}{2}n^3 + 2n^2 + \frac{1}{2}n$	$3n^2$	$O(n^3)$	$O(n^2)$
`trmm`	n^3	$3n^2$	$O(n^3)$	$O(n^2)$
`cholesky`	$\frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$	n^2	$O(n^3)$	$O(n^2)$
`durbin`	$2n^2 + 3n - 5$	$2n$	$O(n^2)$	$O(n)$
`gramschmidt`	$2n^3 + n^2 + n$	$3n^2$	$O(n^3)$	$O(n^2)$
`lu`	$\frac{4}{3}n^3 - \frac{3}{2}n^2 + \frac{1}{6}n$	n^2	$O(n^3)$	$O(n^2)$
`ludcmp`	$\frac{4}{3}n^3 + \frac{1}{2}n^2 - \frac{5}{6}n$	$n^2 + 3n$	$O(n^3)$	$O(n^2)$
`trisolv`	n^2	$n^2 + 2n$	$O(n^2)$	$O(n^2)$
`deriche`	$32n^2$	$4n^2$	$O(n^2)$	$O(n^2)$
`floyd-warshall`	$2n^3$	n^2	$O(n^3)$	$O(n^2)$
`nussinov`	$\frac{1}{6}n^3 - \frac{1}{2}n^2 + \frac{1}{3}n$	$n^2 + n$	$O(n^3)$	$O(n^2)$
`adi`	$30n^3$	$4n^2$	$O(n^3)$	$O(n^2)$
`fdtd-2d`	$11n^3$	$3n^2 + n$	$O(n^3)$	$O(n^2)$
`heat-3d`	$30n^4$	$2n^3$	$O(n^4)$	$O(n^3)$
`jacobi-1d`	$6n^2$	$2n$	$O(n^2)$	$O(n)$
`jacobi-2d`	$10n^3$	$2n^2$	$O(n^3)$	$O(n^2)$
`seidel-2d`	$9n^3$	n^2	$O(n^3)$	$O(n^2)$