# Optimising Matrix-Matrix Multiplications

pbqk24

14 December 2018

## 1 Disclaimer

This work made use of the facilities of the Hamilton HPC Service of Durham University. Unless otherwise stated all performance measures and data presented were performed/gathered on a node in the par7 queue on Hamilton HPC Service, utilizing 1 CPU core with exclusive access, with binaries compiled with gcc.

## 2 Sparse Matrix-Matrix Multiplication

### 2.1 Basic Sparse Implementation

The provided basic implementation of sparse-sparse was initially profiled using gprof in order to determine the hotspots to target for improvement. This was done for all the small matrices. This analysis showed that nearly all the processing time was spent in the naive dense implementation of multiplication, and thus this was the focus of improvement. The timings produced by this analysis is shown in table 1.

### 2.2 Optimised Sparse Implementation

The initial sparse-sparse multiplication was implemented for COO-formatted matrices. This worked by looping over each coordinate in the first matrix (A), and checking each coordinate in the second matrix (B) to see if they need to be multiplied together (i.e. coordinates are (a,b) and (b,c)). As this searched the whole list of coordinates in B for each coordinate in A, it was quite slow, although still yielded a large improvement over the basic implementation for some matrices. This implementation was then improved by sorting the order of coordinates stored in A and B, and utilizing this to reduce the number of coordinates in B that were queried for each coordinate in A by allowing the search through B to be stopped early. The gprof profiling of both of these implementations are shown in table 1 below.

| Matrices Used | Basic Implementation | Initial Implementation | Optimised Implementation |
|---|---|---|---|
| DG1 | 3.29 | 0.62 | 0.49 |
| DG2 | 34.73 | 16.82 | 13.88 |
| DG3 | 163.42 | 209.93 | 167.03 |
| DG4 | 561.14 | 1242.69 | 266.02 |
| Small-CG1 | 5.50 | 131.76 | 101.02 |
| Small-CG2 | 4.96 | 10.42 | 8.30 |

Table 1: Gprof run times for small matrices, for the basic sparse-sparse multiplication implementation, the initial improved implementation, and the optimised implementation

In order to determine how to further optimise the algorithm these three implementations were also profiled using Intel Advisor to produce a roofline plot. These plots are presented below.

### 2.3 Multiplication of Sums of Sparse Matrices

The multiplication of sums of sparse matrices funtion was implemented in two steps. First, the two sets of sparse matrices are added together, and then the resulting matrices are given as input to the sparse matrix multiplication function discussed above. The summing is done by first sorting

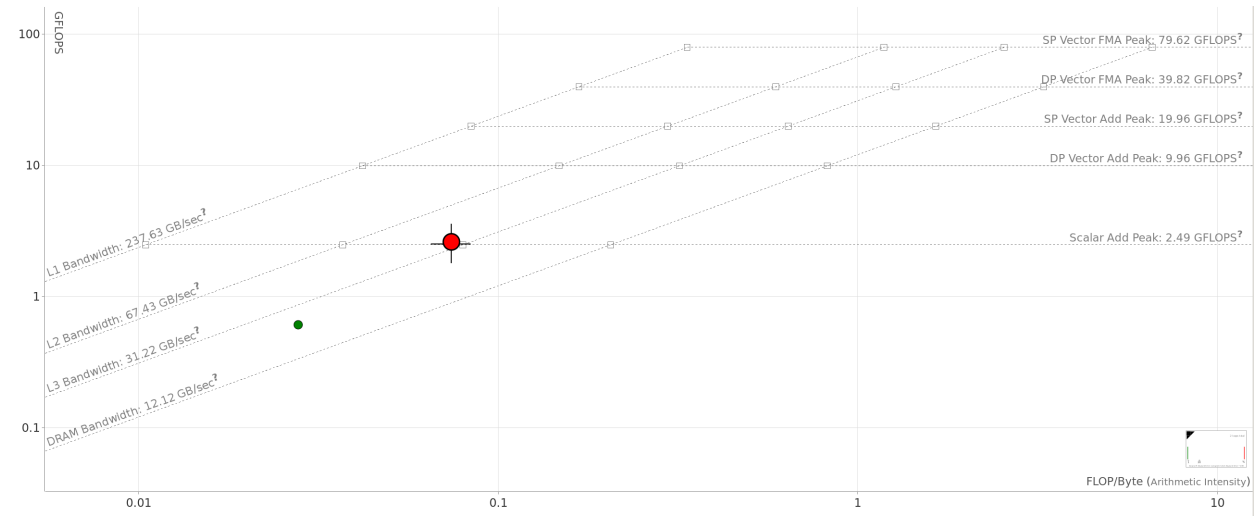


Figure 1: Roofline plot of the basic sparse-sparse matrix multiplication implementation. The algorithm was extensively vectorised automatically by the compiler, causing it to be memory-bound near L3 bandwidth with a performance of 2.62 GFLOP/s and arithmetic intensity of 0.074 FLOP/Byte.

the input matrices, and then merging them similar to merge-sort: the lowest index of both the lists (of coordinates) are compared, if the coordinates are the same they are summed and added as one coordinate to the output. Otherwise, the coordinate with the lower row value, or the lower column value if the row values are the same, is added to the output. This is repeated until one list is empty, at which point all the coordinates of the other list are added. This is performed twice to add the three matrices together for each input (i.e. $A+B+C$ and $D+E+F$), and then input to the multiplication.

### 2.4 Optimisation through Vectorisation

As discussed in the roofline plots in figures 1, 2, and 3, both the initial implementation and the optimised implementation were vectorised by the compiler. Due to the low arithmetic intensity of the initial implementation, and the design of the optimised one, any further vectorisation would result in minimal, if any, improvements to the performance. In order to achieve higher performance in sparse-sparse matrix multiplication, it would be better to improve the underlying algorithm used. For example, this could be adapted to convert the COO matrices to CSR formatted ones, i.e. Compressed Sparse Row format. Through this, the arithmetic intensity could be massively increased by reducing the number of coordinates read and analyzed in B for each coordinate in A to a minimum. This would yield great performance improvements, and allow the implementation to benefit more from vectorisation. The principle of this is shown in figure 4. In addition to this, performance could be further improved by adding a heuristic to determine when it might be better to use a dense multiplication approach, for example when the number of non-zeros in A or B is greater than a quarter of the maximum. This could lead to massively improved performance in certain cases.

## 3 Dense Matrix-Matrix Multiplication with BLIS

### 3.1 Approach and Description of Realisation

The implementation of the BLIS approach for dense matrix-matrix was realised as a sequence of four nested for loops. Each loop in this sequence represents a further sub-division of the input and output matrices: the outer two loops pack B and A into smaller data structures with different ordering, while the inner two loops further extract a smaller row/column of values from these data structures. This is done in such a way that inside the innermost for loop the micro kernel function can be applied to multiply a row from the packed area of A by a column from the packed area of B, and add the output to the correct segment of C (the output matrix). This concept is illustrated in figure 7.

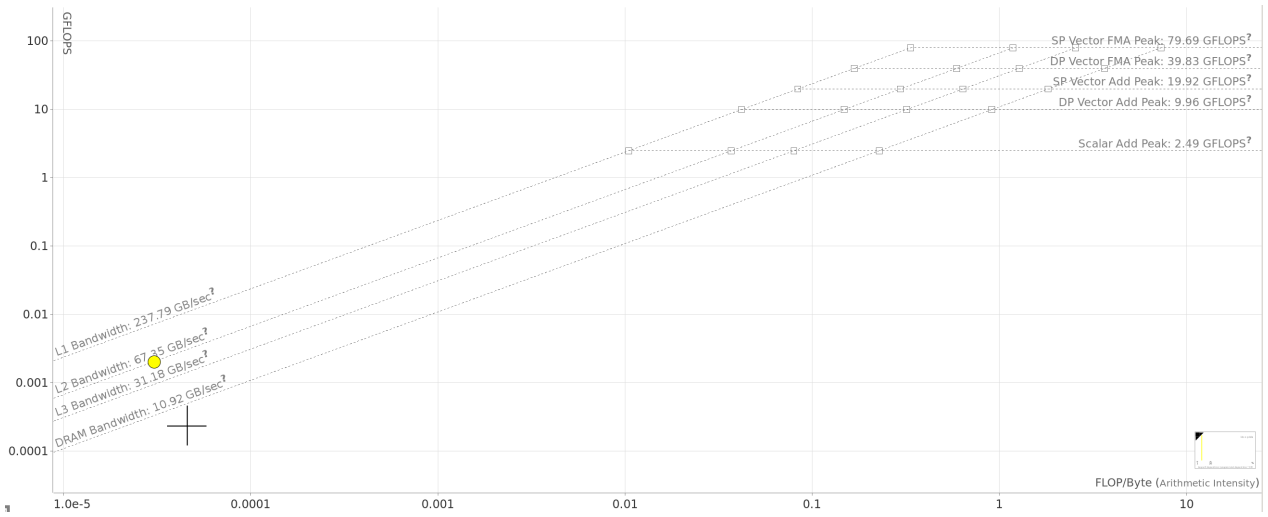


Figure 2: Roofline plot of the initial sparse-sparse matrix multiplication implementation. The algorithm was vectorised by the compiler, resulting in near L2 bandwidth with a performance of 0.002 GFLOP/s and arithmetic intensity of 0.000031 FLOP/Byte. Despite this the low arithmetic intensity kept the performance of the implementation low.

The implementation runs for any $m, k, n > 0$. This is achieved by utilizing the basic implementation if $m, n$, and $k$ are all $<= 700$, as this achieves better performance. In addition to this, since the additional parameters $(m_c, k_c, m_r, n_r)$ are set as constants that yield the best performance, any case where the resulting sub-divisions don't fit perfectly are handled by padding the sub-regions with zeros so that they are the full size. By doing this the same micro kernel can be used throughout, and the product added to C as usual.

### 3.2 Performance Analysis

In order to test the performance of the BLIS implementation, it was benchmarked on Hamilton. Figures 5 and 6 show the result of the benchmark of the basic implementation and the BLIS implementation, respectively. The benchmarks were both run for two square matrices (i.e. $m = k = n$, from 20 to 2000 (inclusive) with steps of 20. This was chosen as it tested a mix of small and large matrices, with a small enough step size to be able to see meaningful trends.

In conjunction with this benchmark testing, the peak performance of a node in the par7 queue on Hamilton was tested. This was run with the same settings as the benchmark testing, and produced an average peak of 80 GFLOP/s with a problem size of 1000. Thus, the BLIS implementation achieves roughly 7.5% of peak on this hardware.

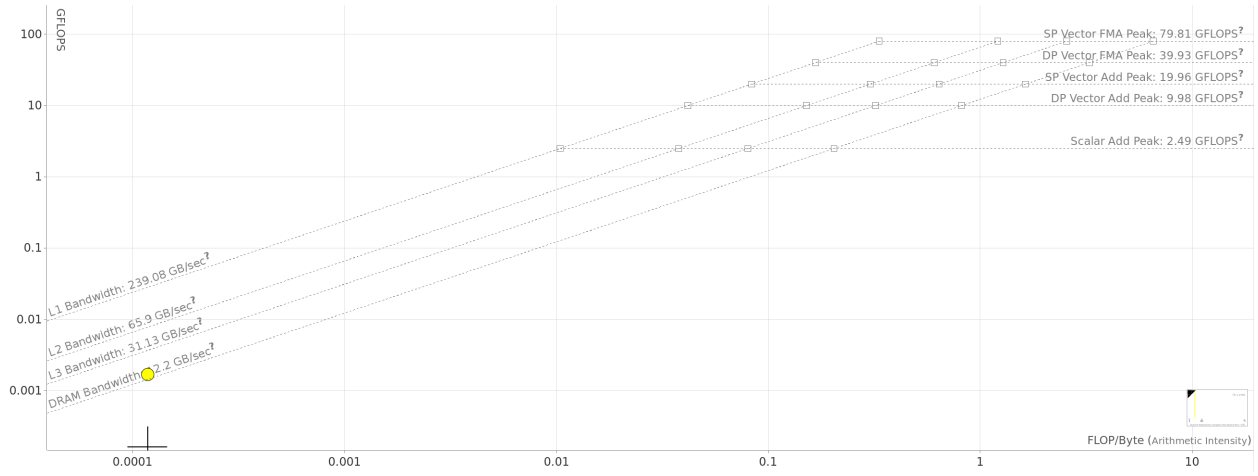


Figure 3: Roofline plot of the the optimised sparse-sparse matrix multiplication implementation. The algorithm was slightly vectorised automatically by the compiler, bringing it to DRAM bandwidth with a performance of 0.0017 GFLOP/s and arithmetic intensity of 0.00012 FLOP/Byte. Due to the algorithmic improvements to the implementation, the arithmetic intensity improved drastically from the initial implementation. However, this also reduced the level of vectorisation that could be performed by the compiler. Overall, the performance improved from the initial implementation

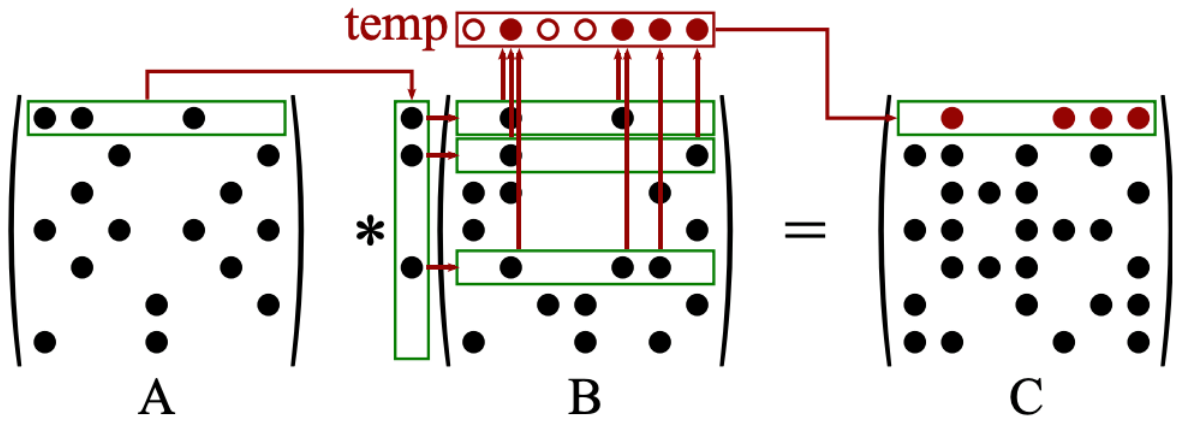


Figure 4: The efficiency of CSR-CSR sparse matrix multiplication. For each row in A, each non-zero entry marks a row in B that may contain relevant non-zero entries. As both matrices are stored in CSR format, it is easy to loop over and read the rows of each matrix, and the number of irrelevant entries read while searching for relevant ones is minimised. Source: https://archive.org/details/arxiv-1303.1651

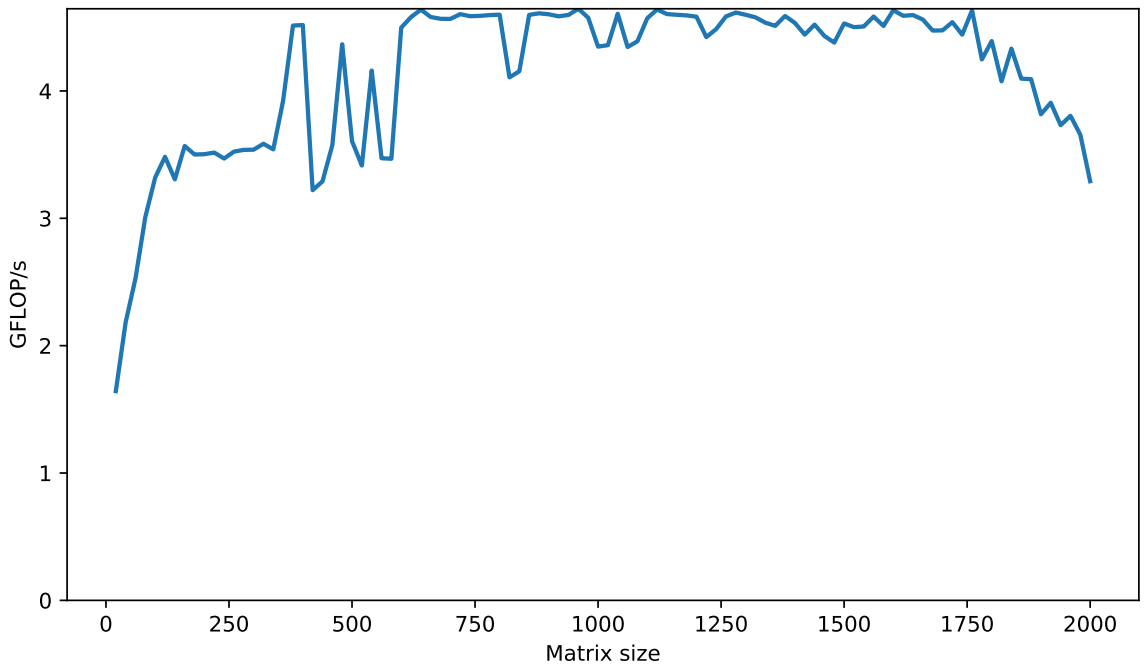


Figure 5: Benchmark plot for basic dense matrix-matrix multiplication. The highest performance attained is around 4.5 GFLOP/s, achieved for matrix sizes 640 to 1760, after which the performance starts to drop.

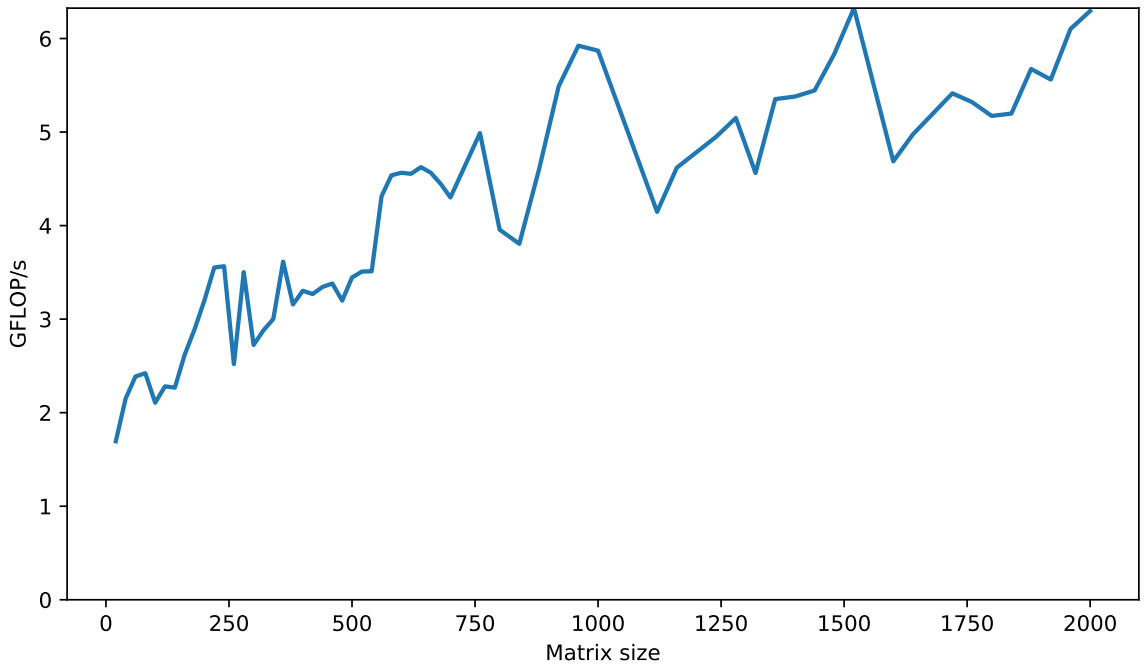


Figure 6: Benchmark plot for BLIS implementation. The highest performance achieved is around 6.2 GFLOP/s, for matrix size 2000. The performance grows steadily from a size of 20 to the largest size tested, 2000.

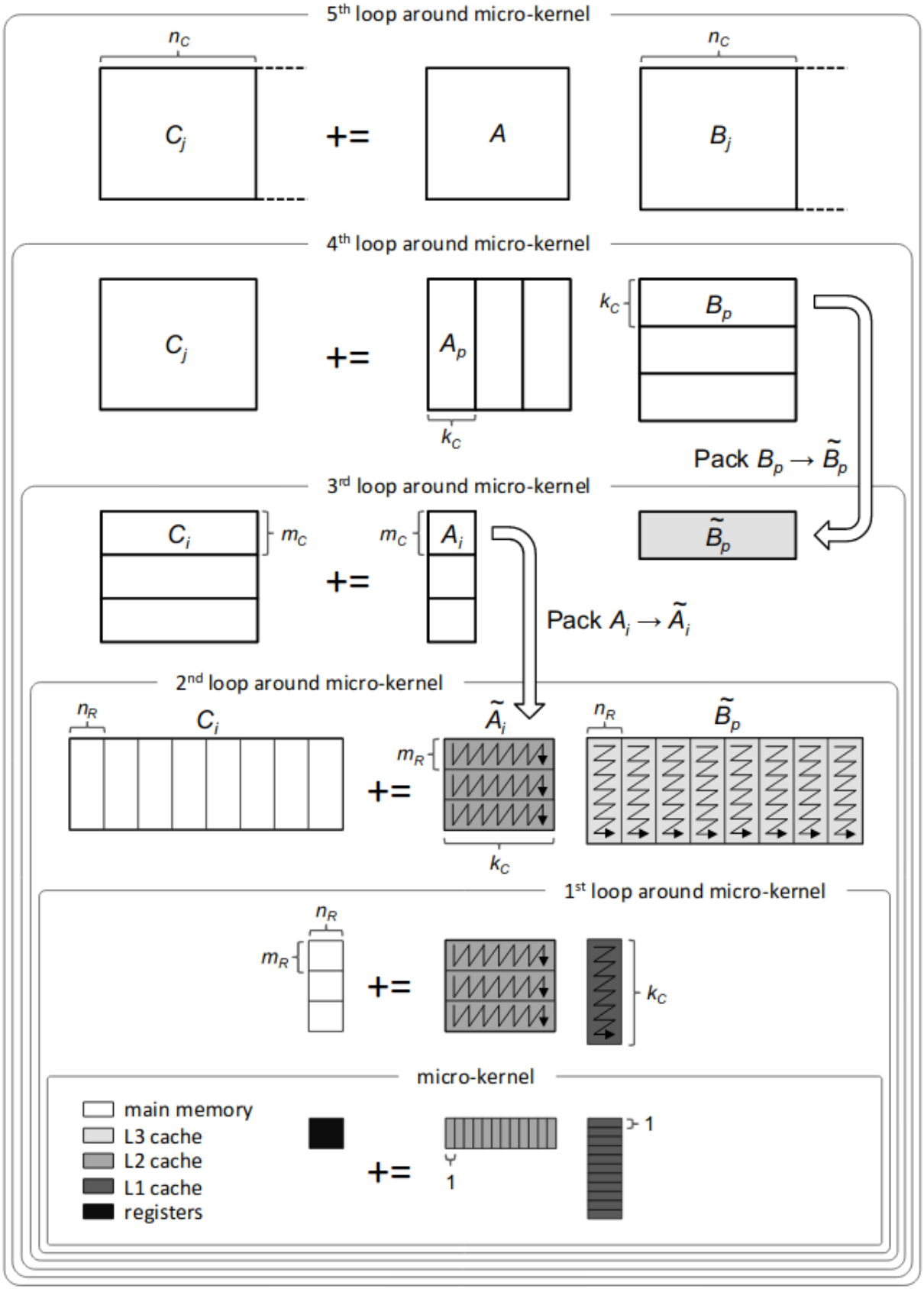


Figure 7: Outline of the processing of input in BLIS. The computation of $C = A * B$ is broken down into many smaller problems. The extra parameters $m_c, k_c, m_r, n_r$ are used.
Source: https://www.cs.utexas.edu/users/flame/pubs/blis6_toms_rev0.pdf