Machine Balance

$$B_m = \frac{\text{memory bandwidth(Gwords/s)}}{\text{peak performance (Gflops/s)}} = \frac{b_{\max}}{P_{\max}}$$

- Units: words/flop
- Ideally, $B_m \approx 1$, but usually $B_m \ll 1$, and the trend is that B_m is further decreasing
- Typical value:

$$\frac{(10\ \text{GB/s})\ /\ (8\ \text{B/word})}{20\ \text{Gflops/s}} = 0.06\ \text{words/flop}$$

Code Balance

$$B_c = \frac{\text{data traffic (words)}}{\text{floating point ops (flops)}}$$

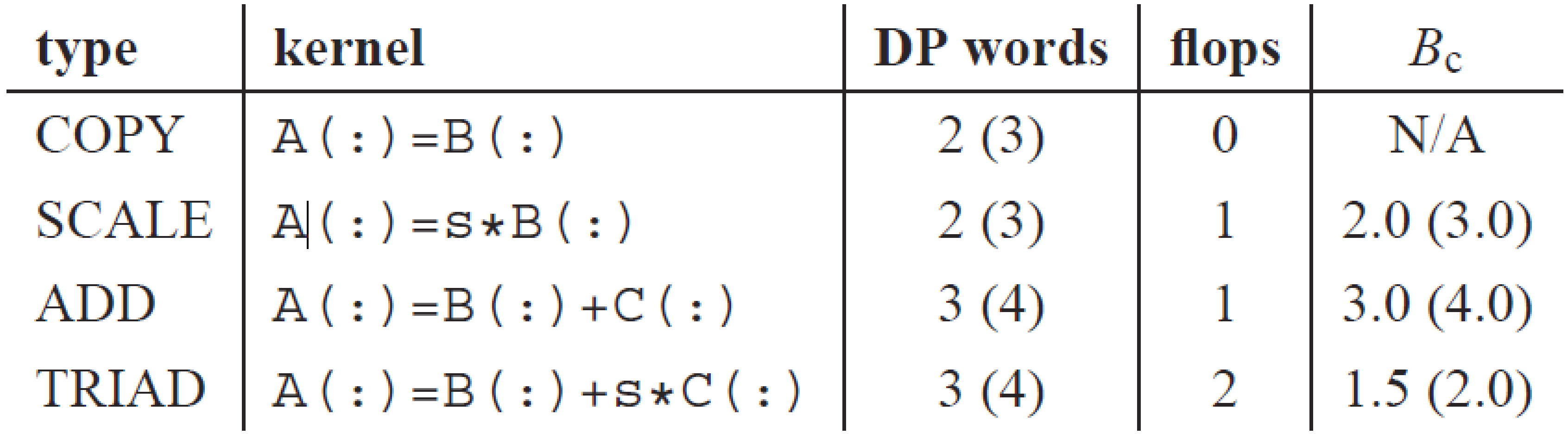
$1/B_c$ = computational intensity (flops/word)

B_c/B_m = fraction of peak flops

If $B_m < B_c$, then the machine cannot satisfy the data traffic requirements (performance is limited by memory bandwidth)

What is B_c for: a[i]=b[i]+c[i] ?

STREAM Benchmark

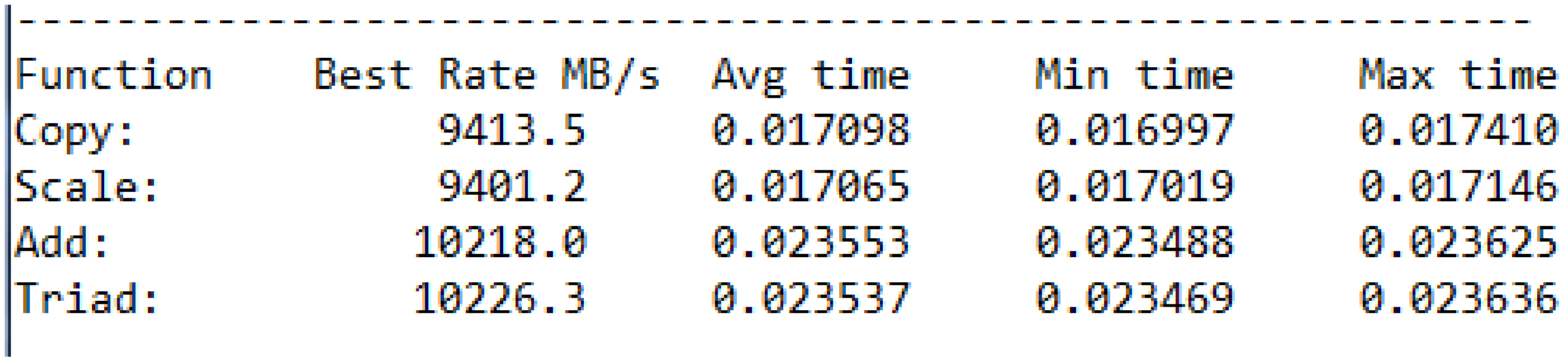


type	kernel	DP words	flops	B_c
COPY	`A(:)=B(:)`	2 (3)	0	N/A
SCALE	`A(:)=s*B(:)`	2 (3)	1	2.0 (3.0)
ADD	`A(:)=B(:)+C(:)`	3 (4)	1	3.0 (4.0)
TRIAD	`A(:)=B(:)+s*C(:)`	3 (4)	2	1.5 (2.0)

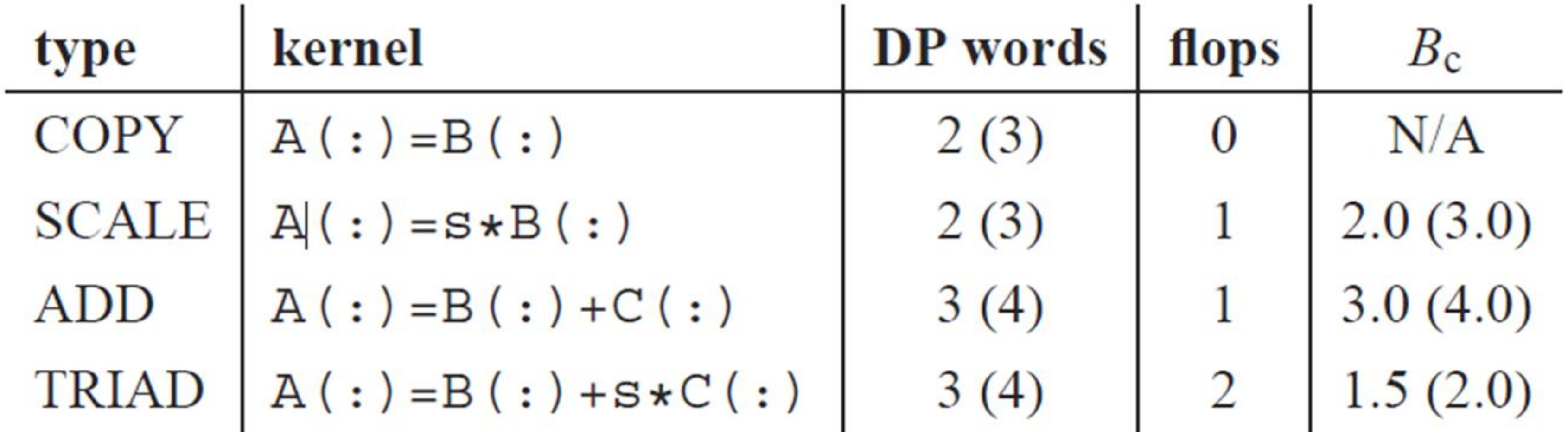
- Stream benchmark bandwidth b_{stream} reflects the true capabilities of the hardware
- Use b_{stream} rather than theoretical b_{max} in performance models

Bracketed numbers assume two memory accesses for a write (for write-allocate caches)

Stream benchmark output



```
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           9413.5     0.017098     0.016997     0.017410
Scale:          9401.2     0.017065     0.017019     0.017146
Add:           10218.0     0.023553     0.023488     0.023625
Triad:         10226.3     0.023537     0.023469     0.023636
```



type	kernel	DP words	flops	B_c
COPY	A(:)=B(:)	2 (3)	0	N/A
SCALE	A(:)=s⋆B(:)	2 (3)	1	2.0 (3.0)
ADD	A(:)=B(:)+C(:)	3 (4)	1	3.0 (4.0)
TRIAD	A(:)=B(:)+s⋆C(:)	3 (4)	2	1.5 (2.0)

Writes are somewhat more expensive than reads (possibly due to lower write bandwidth compared to read bandwidth,and/or write-allocate caches).

Cache lines

- Cache is organized into chunks (of typically 64 bytes) called *cache lines*
- This is because it is inefficient to tag single bytes or words in cache (i.e., the address of the word in cache), and inefficient to read bytes or words from memory one at a time (cost of reading 64 bytes is about the same as reading 1 byte)
- Think of memory as being organized into 64 byte chunks. When a word is read from memory into cache, the entire chunk that contains the chunk is read into a cache line
- Many optimizations are based on trying to efficiently use all the elements of a cache line when it is read from memory

Matrix transposition and cache lines

- Pseudocode (doesn't show how data is laid out, but let's specify row-major ordering)

```
loop i = 1 to n
  loop j = 1 to n
    b(i,j) = a(j,i)
  endloop
endloop
```

- Access is consecutive in one matrix and strided in other matrix; this is a problem when both matrices do not fit into cache
- For large n, code can be rewritten to more efficiently use all the elements of a cache line

Prefetching

- Main memory latency is hundreds of cycles. If we knew what data in memory we needed beforehand, this data can be prefetched. Prefetching is needed to achieve the full memory bandwidth capable on a machine.
- Hardware prefetching
  - Prefetching is triggered by two or more consecutive cache misses in succeeding or preceding cache lines (not necessarily unit stride)
  - Prefetching can be triggered when you don't want it, causing "good" data to be evicted from cache. In this case, prefetching can often be turned off by setting hardware registers
- Software prefetching
  - Programmer can insert __mm_prefetch(addr, hint) intrinsic to manually prefetch data, where addr is a pointer, and hint specifies, e.g., which levels of cache to load into

Matrix-vector product and prefetching

c = A b

- How should the matrix A be stored in order to access memory consecutively?
- For large matrices, how should the matrix A be stored to reduce number of times vector b is read?

Matrix multiplication

- Naïve algorithm (both matrices in column-major ordering)
- Optimization 1: transpose first matrix
  - Advantage: streaming access of both matrices
  - Disadvantage: need additional copy of matrix
- Optimization 2: blocking (tiling)
  - Efficient use of cache lines for large matrices
  - More complicated than naïve algorithm

Question

- What is the “computational intensity” (inverse of “code balance”) for matrix-matrix multiplication? For C=AB, assume A and B are n-by-n matrices.

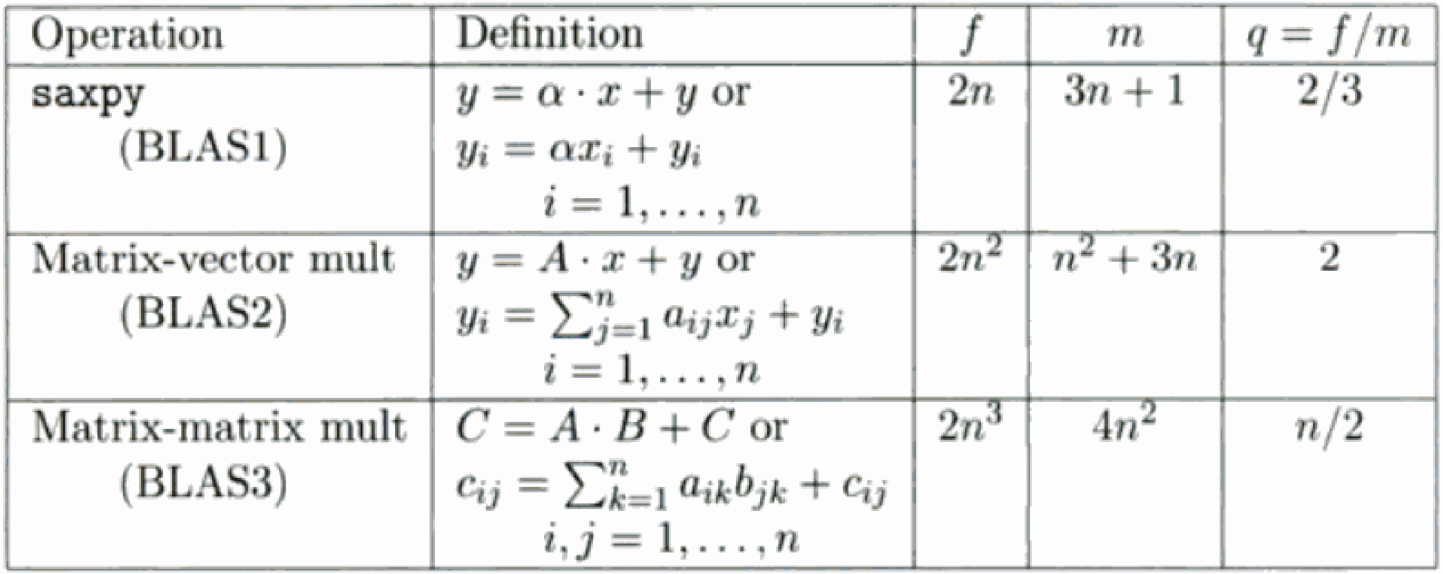
Linear algebra operations

- Vector addition
- Matrix-vector multiplication
- Matrix multiplication

- These are common kernels in scientific computing. They are not easy to optimize, but optimized codes are available.

BLAS: Basic Linear Algebra Subroutines

Floating point operations per memory access



Operation	Definition	f	m	$q = f/m$
saxpy (BLAS1)	$y = \alpha \cdot x + y$ or $y_i = \alpha x_i + y_i$ $i = 1, \ldots, n$	$2n$	$3n+1$	$2/3$
Matrix-vector mult (BLAS2)	$y = A \cdot x + y$ or $y_i = \sum_{j=1}^{n} a_{ij} x_j + y_i$ $i = 1, \ldots, n$	$2n^2$	$n^2 + 3n$	2
Matrix-matrix mult (BLAS3)	$C = A \cdot B + C$ or $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{jk} + c_{ij}$ $i, j = 1, \ldots, n$	$2n^3$	$4n^2$	$n/2$

Demmel

Strive to build algorithms with higher-level BLAS, e.g., BLAS3, which was motivated by cache-based computers

BLAS: Basic Linear Algebra Subroutines

  - Level 1 BLAS: scale, saxpy, dot product, norms
  - Level 2 BLAS: sgemv (matrix-vector), rank 1 updates, rank 2 updates, triangular matvecs, triangular solves
  - Level 3 BLAS: matrix-matrix product, rank-k updates, triangular solves with multiple rhs
- Optimized implementations for each platform
  - various vendors (ACML, ESSL, Intel MKL)
  - GotoBLAS (up to Nehalem), OpenBLAS
  - ATLAS (autotuning)
  - Reference BLAS from netlib (not optimized)

BLAS is everywhere

- Matrix multiplication kernel is called DGEMM
- DGEMM can achieve > 90% of peak flop rate

- http://vergil.chemistry.gatech.edu/news/hexagon.html

FMA: Fused multiply-add instruction

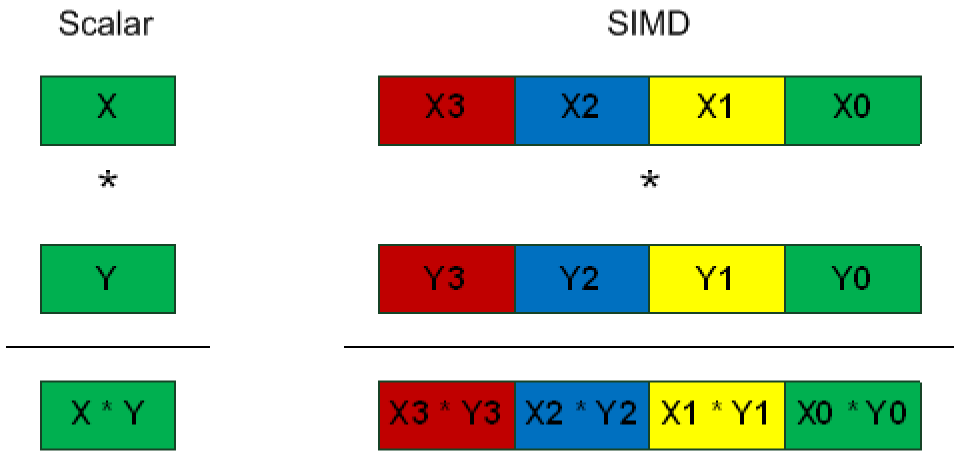
- multiply-adds are common, e.g., in DGEMM

```
z1 = x1*y1 + x2*y2 + ...
```

- doubles the peak flop rate
- also important: the multiply-add is "fused":
  - result of multiply is effectively not rounded before addition is performed
- FMA instructions have been available for a long time on IBM POWER and other chips. Only recently they are widely available on Intel chips in the Haswell architecture.

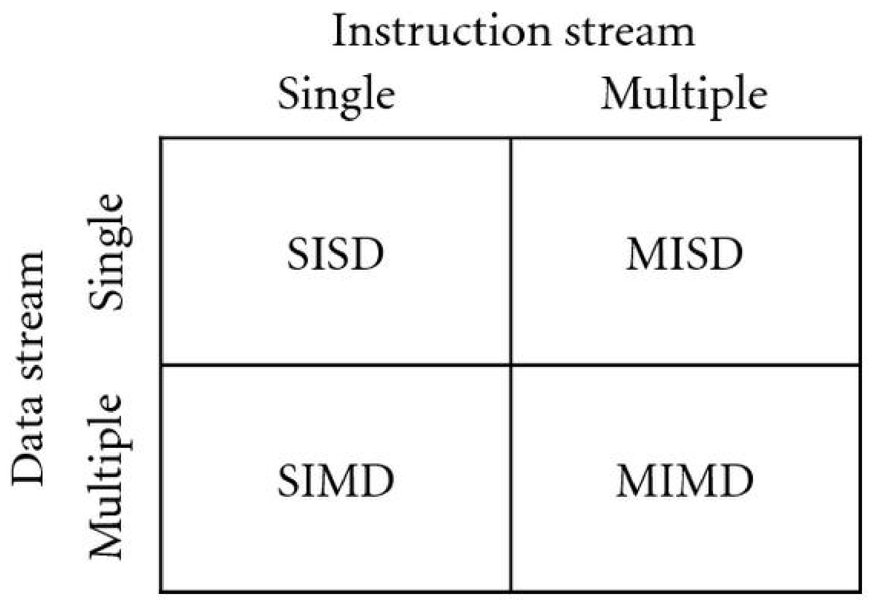
Processor-level SIMD

- SIMD instruction can perform an operation on multiple words simultaneously



- This is a form of data parallelism. One version is called “Streaming SIMD Extensions” (SSE)

Flynn's Taxonomy



		Instruction stream	
		Single	Multiple
Data stream	Single	SISD	MISD
	Multiple	SIMD	MIMD

- SIMD: single-instruction, multiple data
- Many earlier computers fall under the SIMD model or have SIMD capabilities, e.g., CRAY vector supercomputers

Processor-level SIMD

- If your code is memory bandwidth bound, then SIMD generally will not help improve performance
- Using SIMD instructions can shorten your code (fewer instructions to do the same thing) which might reduce memory bandwidth for instructions

Recent SIMD versions

• Nehalem:	SSE 4.2 (128 bit)
• Sandy Bridge:	AVX (256 bit)
• Haswell:	AVX2 (256 bit with FMA)
• MIC:	AVX-512 (512 bit)

Versions are not backward compatible, i.e., cannot use AVX instructions on Nehalem.

How to exploit SIMD

- Compiler automatically generates SIMD instructions (auto vectorization) if compiling with –O3. This is usually the best option.
- Programmer inserts assembly code.
- Programmer inserts “intrinsics” (which map to assembly instructions, but you can use pointers rather than registers). Also need to tell the compiler how to map the instrinics, e.g., –msse4.2 with gcc on Nehalem

Using SIMD intrinsics (SSE)

- 128-bit registers have type __m128 (single precision), __m128d (double precision)
- Many instructions, including: _mm_load_ps, _mm_add_ps
- Data needs to be aligned to use most load/store instructions, use posix_memalign

- For AVX, we have __m256, _mm256_add_ps, etc.

sse1.c example

- sse1.c example computes vector sum a = a + b using SSE (code is in repo)
- Things to try:
  - Unalign the arrays so compiler cannot generate SIMD code, and SIMD code cannot load aligned
  - Change length of loop so that we no longer operate in L1 cache
  - Unroll loops?
  - Compile with different optimization levels?
  - Compile with –S to look at the assembly code

Intel compiler optimization reports

- icc … -qopt-report=[0-5]
  icc … -qopt-report-phase=vec

  compiler will output an optimization report in the file xx.optrpt