Basic Performance Analysis

Given that we are assuming a hierarchical memory and a combined floating point multiply/add instruction that computes 2 operations per cycle we would expect that the best computational rate we can have per processor is $R_{max} = 2$ assuming all data is in the local memory or registers that can keep up with the CPU.

Of course we assume that data starts out in the main memory and therefore the computational rate must be scaled by the number of transfers from main memory to local memory or registers. Below we analyze the data movement for the BLAS and motivate the use of BLAS2 and BLAS3 by architectural considerations.
Consider the execution of a triad on a register-based vector processor i.e., assume the local memory is actually a vector register set.

Let $x_i, y_i \in \mathbb{R}^L$ where $L$ is the register length and $x = (x_1, \ldots, x_k)^T$ and $y = (y_1, \ldots, y_k)^T$ with $n = kL$. The computation is given by

$$x_i = x_i + \alpha y_i$$

for $i = 1, \ldots, k$.

If V1, V2, V3, V4 are registers (or arrays in local memory) of length L and we abuse notation by assuming the mathematical objects of $x$, $y$ and $\alpha$ can be used to refer to data structures in memory then we have the following pseudocode:

```plaintext
DO i = 1, K
    V1 ← x_i
    V2 ← y_i
    V3 ← \alpha \cdot V2
    V4 ← V3 + V1
    x_i ← V4
END DO
```
Assume that the multiply and add of the vectors can be done in a combined fashion so the computation of V3 and V4 are overlapped (this is called chaining)

Assume that only one vector load or store can occur between main and registers (local memory) at any given time but that such a load can overlap with chained operations

The timing diagram looks like

```
load v1   load v2   store v4
PORT:     |---------| |---------| |---------|
          
*:        |         | v2*alpha |
          |+v1 -> v4 |
+:        |---------|
```

Note that the time is essentially the time it takes to perform a load of a vector of length L followed by a second load and finally a store, i.e., a LOAD-LOAD-STORE transaction where the operands start in the farthest part of memory. Therefore performance is limited by how fast this LLS transaction can be done.
We have $\mu_v = \mu_{min} \approx 3/2$ and the number of operations per main memory transfer is 2/3. To convert this to computations per cycle we must scale by the cycles per memory transfer, $\phi > 1$, from main memory to local memory or register (which is an architectural parameter that can vary) i.e., $R_v = 2/3\phi$ Since there is only one memory port, this is the best we can do. It is clear that this primitive, though it can certainly be used as a total primitive on this type of machine, does not effectively utilize the resources of the processor.
Dotproduct

\[ \gamma \leftarrow \gamma + x^T y = \gamma + \sum_{i=1}^{n} \xi_i \eta_i \]

where \( x, y \in \mathbb{R}^n \).

Suppose we have a vector instruction DOTPROD that can take the dotproduct of two vectors in register or local memory (chaining the multiply and add as before) then the dotproduct pseudocode on one processor is (F1 is a scalar local memory or register):

\[
\begin{align*}
F1 &\rightarrow \gamma \\
\text{DO } i &= 1, K \\
&\quad V1 \leftarrow x_i \\
&\quad V2 \leftarrow y_i \\
&\quad F1 \leftarrow F1 + \text{DOT}(V1, V2) \\
\text{END DO}
\end{align*}
\]
The timing diagram looks like

```
PORT: |--------|--------|--------|
    |        |        |        |
*: |        |        | dot(v1,v2) |
+: |        | +f1 -> f1 |--------|
```

Note that this is a LOAD-LOAD sequence due to the reduction nature of the computation rather than the LOAD-LOAD-STORE of the triad. Therefore, if the memory system can keep up this operation is preferred to the triad on machines with one port.
• The previous discussion has simply moved the primitive from an algorithm to an instruction. We now assume that the DOTPROD instruction exists, is efficient and has no interesting problems. This is not always true.

• On some vector machines there is a significant startup difference between the DOTPROD and other simpler vector instructions.

• On a RISC vector machine doing a reduction operation in general must be carefully considered relative to the hardware available and low level control details.

• The essential point is that the BLAS1 reduction made better use of the limited memory ports (in this case 1) than the BLAS1 nonreduction operation $- \mu_v = 1$ and $R = \phi$. (It is still not as good as we would like to be.)
BLAS2

Motivation:

- efficient register and memory port usage
- 2-D parallelism

Matrix-vector operations: result is a matrix (nonreduction) or a vector (reduction)

Examples:

Rank-1 update (nonreduction):

\[ A \leftarrow A + xy^T \]

Matrix-vector product (reduction):

\[ y \leftarrow y + Ax \]
**Rank-1 update**

Let \( A \in \mathbb{R}^{n_1 \times n_2}, x \in \mathbb{R}^{n_1}, \) and \( y \in \mathbb{R}^{n_2} \)

\[
A \leftarrow A + xy^T
\]

This is equivalent to:

```
do i = 1, n_1
do j = 1, n_2
    \alpha_{ij} = \alpha_{ij} + \xi_i \eta_j
end do
end do
```

where \( A = [\alpha_{ij}], x = (\xi_1, \ldots, \xi_{n_1})^T, \) and \( y = (\eta_1, \ldots, \eta_{n_2})^T. \)

It follows immediately from this definition that

\[
\begin{align*}
\Omega &= 2n_1n_2 \\
\delta &= n_1n_2 + n_1 + n_2 \\
\Theta_{\text{min}} &= 2n_1n_2 + n_1 + n_2 \\
\pi_{\text{max}} &= n_1n_2 \\
\pi_{\text{ave}} &= n_1n_2 \\
\mu_{\text{min}} &= 1 + 1/(2n_1) + 1/(2n_2)
\end{align*}
\]
Consider the rich structure of this operation ($e_i$ is a vector with a 1 in the i-th position and 0 elsewhere)

- $n_2$ column triads of length $n_1$, i.e. $n_2$ parallel BLAS1 operations
  $$(Ae_j) \leftarrow (Ae_j) + \eta_jx$$

- $n_1$ row triads of length $n_2$ (row access required)
  $$(e_j^TA) \leftarrow (e_j^TA) + \xi_iy^T$$
\( k_1k_2 \) rank-1 updates of size \( m_1 \times m_2 \) where \( m_1k_1 = n_1 \) and \( m_2k_2 = n_2 \). (Exact integer multiples are assumed for simplicity.) Partition \( A, x \) and \( y \) as
\[
\begin{pmatrix}
A_{11} & \cdots & A_{1k_2} \\
\vdots & \ddots & \vdots \\
A_{k_11} & \cdots & A_{k_1k_2}
\end{pmatrix}
\]
\( x = (x_1, \ldots, x_{k_1})^T \), and \( y = (y_1, \ldots, y_{k_2})^T \) where \( A_{ij} \in \mathbb{R}^{m_1 \times m_2} \), \( x_i \in \mathbb{R}^{m_1} \) and \( y_i \in \mathbb{R}^{m_2} \).

Then the rank-1 is just
\[
A_{ij} \leftarrow A_{ij} + x_i y_j^T,
\]
for \( i = 1, \ldots, k_1 \) and \( j = 1, \ldots, k_2 \).
Consider the value of $\mu$ achieved for each of the forms mentioned above and compare to $\mu_{\text{min}}$.

- scalar form: can be run on up to $n_1n_2$ processors, each basic task is a multiplication followed by an add. 3 reads, 1 write for every 2 operations. Therefore,

$$\mu_{\text{scalar}} = 2$$

- column triad form: if the basic computational task sent to each processor is a triad of length $n_1$ then we can use up to $n_2$ processors. Reads = $n_2(2n_1 + 1)$, writes = $n_1n_2$. Therefore,

$$\mu_{\text{triad}} = \frac{3}{2} + \frac{1}{2n_1}$$
Note that $\mu_{\text{scalar}}$ and $\mu_{\text{triad}}$ are both on the order of BLAS1 level operations as expected. $\mu_{\text{min}}$ however says we should be able to do better. If we consider the form where the rank-1 is decomposed into $k_1 k_2$ smaller rank-1 updates the situation can improve considerably.

We will choose $m_1$ and $m_2$ so that each smaller rank-1 has a local $\mu$ equal to $\mu_{\text{min}}$ for that size problem, i.e.,

$$\mu_{\text{local}} = 1 + \frac{1}{2m_1} + \frac{1}{2m_2}$$
Fact: If $m_1 \leq L$ or $m_2 \leq L$, where $L$ is the register length, then $A_{ij} + x_i y_j^T$ can be computed with optimal local $\mu$.

We prove this by constructing the code executed on each processor for each of the above conditions.

Assume $m_1 \leq L$ and that the column triad form is to be used on each processor to compute $A_{ij} + x_i y_j^T$. Let $a^{(k)}$ denote the $k$-th column of $A_{ij}$ and let $y_j = (\eta_1, \ldots, \eta_{m_2})^T$. Note that $x_i \in \mathbb{R}^{m_1}$ is used in each of the triads and since $m_1 \leq L$ it can be loaded into a vector register and kept there for the entire small rank-1 update. Therefore, the key to achieving minimal $\mu$ locally is operand reuse.
The pseudocode is

\[ V_1 \leftarrow x_i \]
\[ \text{do } k=1, m_2 \]
\[ F_0 \leftarrow \eta_k \]
\[ V_2 \leftarrow a^{(k)} \]
\[ V_3 \leftarrow V_1 \times F_0 \]
\[ V_2 \leftarrow V_3 + V_2 \]
\[ a^{(k)} \leftarrow V_2 \]
\[ \text{end do} \]
Note that we have achieved optimality locally by considering the decomposition into several BLAS1 level kernels. This was not done at a global level but at the appropriate level in this case after decomposition into smaller rank-1 updates.

This also represents the first example of a constrained block size optimization problem albeit a simple one. The task was to minimize $\mu_{local}$ given that you had local storage (a vector register) with a particular size and shape, a $1 \times L$ array of locations.

Also note that this is an example of the fact that optimization of performance often requires looking at the interaction with respect to some metric across a series of BLAS primitives at the same level that together make up a primitive or algorithm of higher complexity. In this case, a series of triads were optimized with respect to their operand use (common operands were kept in local memory) to make an efficient rank-1 update on a single processor.

This type of analysis occurs, at various levels of complexity, throughout high performance algorithm design.
The global value of $\mu$ can be found by multiplying the number of transfers by $k_1 k_2$ and the number of operations by $k_1 k_2$ (since there are $k_1 k_2$ smaller rank-1 updates performed). Since this is just multiplying the numerator and denominator of $\mu_{local}$ by the same constant the global value of $\mu$ for the rank-1 is

$$\mu_{r1}(m_1, m_2) = 1 + \frac{1}{2m_1} + \frac{1}{2m_2}$$

If the number of iterations is taken to be $p = k_1 k_2$ then this becomes

$$\mu_{p}(m_1, m_2) = 1 + \frac{p}{2n_1 n_2} (m_1 + m_2)$$

($p$ will later be taken to be the number of processors)

Note that this value of $\mu$ is consistent with the previous observations made:

- scalar form to each processor implies $m_1 = m_2 = 1$ and $\mu_{p}(1, 1) = 2$.

- column triad form to each processor implies $m_1 = n_1, m_2 = 1$ and $\mu_{p}(n_1, 1) = 3/2 + 1/(2n_1)$
Partitioning choice

The choice of $m_1$ and $m_2$ and the kernel used on each processor, e.g., scalar, row or column triad, depends upon the architecture, its associated costs, and the problem size.

Some general constraints can be stated however. Assume that the column triad form is to be used for each processors computation of $A_{ij} + x_i y_j^T$. (Similar constraints are possible if row triads are used with the roles of $m_1$ and $m_2$ reversed.)

- $m_1 \leq L$ (in practice, the $L$ is often replaced by some small multiple of $L$ due to unrolling)

- $m_1$ large enough to offset vector startup costs (long enough vector lengths to use vector processing on each processor)

- $m_2$ large enough to offset cost of loading $x_i$ into registers (enough reuse per processor to move away from levels of $\mu$ typical for BLAS1 triads)

- Since $k_1 k_2$ is the number of iterations for a single vector processor it tends to be taken so that one $k_i = 1$ and the other is set according to the constraints above. For parallel processors typically $p \approx k_1 k_2$. 
The optimal number of loads and stores are achieved for the problem with dimension $m_1 \times m_2$. To see this note that $x_i$ is loaded once giving $m_1$ reads. Each element of $y$ is read once via the load of $\eta_k$ giving $m_2$ reads. Each column of $A_{ij}$ is loaded and stored once giving $2m_1m_2$ transfers. This gives a total of $2m_1m_2 + m_1 + m_2$ transfers for $2m_1m_2$ operations which is locally optimal as desired.

If $m_2 \leq L$ a similar strategy can be used to achieve a locally optimal version.
Matrix-vector product

Let $A \in \mathbb{R}^{n_1 \times n_2}$, $x \in \mathbb{R}^{n_2}$, and $y \in \mathbb{R}^{n_1}$

$$y \leftarrow y + Ax$$

This is equivalent to:

$$
\begin{align*}
d\text{o } i &= 1, n_1 \\
d\text{o } j &= 1, n_2 \\
\eta_i &= \eta_i + \alpha_{ij} \xi_j \\
\text{end do} \\
\text{end do}
\end{align*}
$$

where $A = [\alpha_{ij}]$, $x = (\xi_1, \ldots, \xi_{n_2})^T$, and $y = (\eta_1, \ldots, \eta_{n_1})^T$. 
It follows immediately from this definition that

\[ \Omega = 2n_1n_2 \]
\[ \delta = n_1n_2 + n_1 + n_2 \]
\[ \Theta_{\text{min}} = n_1n_2 + 2n_1 + n_2 \]
\[ \pi_{\text{max}} = n_1n_2 \]
\[ \pi_{\text{ave}} = (2n_1n_2)/(\log(n_2) + 2) \]
\[ \mu_{\text{min}} = 1/2 + 1/(2n_1) + 1/(n_2) \]

The value of \( \pi_{\text{ave}} \) follows from the fact that each element of \( y \) is the result of a dotproduct for which the log-tree form discussed in the BLAS1 section can be used.

Note that the value of \( \mu_{\text{min}} \) is smaller than the rank-1 and any of the BLAS1 primitives. This is due to the fact that the operation is a reduction.

It is consistent with what we expect from a chained * and + (or parallel functional units).
As with the rank-1, the matrix-vector product has a lot of structure.

- $n_1$ independent dotproducts of length $n_2$: $\eta_i \leftarrow \eta_i + r_i^T x$, $1 \leq i \leq n_1$ where $r_i = e_i^T A \in \mathbb{R}^{n_2}$

- $n_2$ triads each of length $n_1$:

$$y \leftarrow y + \sum_{j=1}^{n_2} \xi_j c_j$$

where $c_j = Ae_j \in \mathbb{R}^{n_1}$.

- As with the rank-1 decomposing the problem into several smaller versions of itself gives us the general form of the primitive. Partition $A$, $x$ and $y$ as

$$
\begin{pmatrix}
A_{11} & \cdots & A_{1k_2} \\
\vdots & \ddots & \vdots \\
A_{k_11} & \cdots & A_{k_1k_2}
\end{pmatrix}
$$

$x = (x_1, \ldots, x_{k_2})^T$, and $y = (y_1, \ldots, y_{k_1})^T$ where $A_{ij} \in \mathbb{R}^{m_1 \times m_2}$, $x_i \in \mathbb{R}^{m_2}$ and $y_i \in \mathbb{R}^{m_1}$, $n_1 = k_1 m_1$, and $n_2 = k_2 m_2$. 
The matrix-vector product is then

$$\text{do } i = 1, k_1$$
$$y_i \leftarrow y_i + A_{i1}x_1 + \cdots + A_{ik_2}x_{k_2}$$
$$\text{end do}$$
Once again consider a smaller problem with optimal $\mu$.

**Fact:** If $m_1 \leq L$ or $m_2 \leq L$ then the local computation of $y_i \leftarrow y_i + A_{ij}x_j$ can be done with locally optimal $\mu(m_1,m_2) = 1/2 + 1/(2m_1) + 1/m_2$.

Assume $m_1 \leq L$ and that a column triad form of the primitive is to be used within the processor. Let $a^{(k)}$ denote the k-th column of $A_{ij}$ and let $x_j = (\xi_1, \ldots, \xi_{m_2})^T$. Note that $y_i \in \mathbb{R}^{m_1}$ is updated by a series of $m_2$ triads. Since $m_1 \leq L$ the update can be accumulated in a vector register without writing $y_i$ to memory after each triad. *Therefore the key to achieving locally optimal $\mu$ is the accumulation of the result of several vector operations in a vector register.*
The pseudocode is as follows:

V1 ← y_i
do k=1, m_2
    F0 ← \xi_k
    V2 ← a^{(k)}
    V2 ← F0*V2
    V1 ← V1 + V2
end do
y_i ← V1
Clearly, each column of $A_{ij}$ is read once, $x_j$ is read once, and $y_i$ is read and written once. This is a total of $m_1m_2 + 2m_1 + m_2$ transfers and a locally optimal $\mu$.

If $m_2 \leq$ then a different form of the local operation must be used to achieve local optimality. The derivation of this form is left as an exercise.
BLAS2 Summary

• trend in preferences is the same as BLAS1, i.e.,
  reduction preferred on a register-based vector pro-
  cessor (or even a small number of them)

• Computations still depend on the memory rate (1
  operation per transfer for rank-1, 2 operations per
  transfer for matrix-vector). They exploit efficient-
  ly the port/register bottleneck on register-based
  single-port processors. If, however, the register-
  s had a bandwidth larger than the ideal memory
  bandwidth these primitives do not have the neces-
  sary data locality to saturate the pipelines.
• The previous discussions tell us that for BLAS2 we still run at the bandwidth of the memory in which the operands start, which for large matrices must be main memory.

• However, at least for matrix vector we have the desired 2 operations per transfer that the \( \mu_{\text{min}} \) predicts.

• It was assumed that the designer was writing at as low a level as needed, e.g., assembler.

• What if you are not writing in assembler? You end up attempting to make the restructuring compiler/code generator give you as close to this kind of code as possible.

• Most restructuring compiler/code generator do a reasonable job on this type of code. (Cray in particular) Sparse matrix codes are a different story we will discuss later.