Embedded Systems

Data Blocking and Coding for Increased Cache Performance

**Goals**

- The goals of a cache aware optimization technique should be to:
  - Minimize the stride by which data is accessed from memory
    - Stride-1 processing would represent the case where data elements were accessed from consecutive, sequential memory locations
    - Stride-2 processing would represent the case where data elements were accessed from every other memory location
    - Stride minimization is a technique for increasing spatial locality of reference
  - Access data in a block-like fashion
    - Blocking (also known as strip mining) is an access technique whereby data is processed in blocks that fit in the cache
    - Accessing data in a blocked fashion is a loose attempt to increase temporal locality of reference
Data Blocking

- The concept of data blocking is simple:
  - process the data in small enough chunks that they fit in the cache
- Two reasons for blocking larger arrays
  - If some arrays must be processed with a large stride
  - If the data values are used many times
    - In this case, you want to do as many computations as possible with the data while it is in the cache and before it is flushed out by more recently accessed lines
    - This applies even if everything is stride 1
    - If neither condition is true, there is no point in blocking

Code Examples

- All stride 1
- No data re-use
- No point in blocking

```c
// All stride one access
// No data reuse
// s represents a scalar quantity
for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
        s = s + a[j][i]*b[j][i];
    }
}
```
Code Examples (continued)

- Unavoidable bad stride
- No data re-use

```c
// Unavoidable bad stride access
// No data reuse
// s represents a scalar quantity
// Access to matrix b is stride one
// Access to matrix a is stride n (if n is the matrix size)
for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
        s = s + a[i][j] * b[j][i];
    }
}
```

Code Examples (continued)

- All stride 1
- Significant data re-use
- Matrix Multiple Transpose (C=ABᵀ)

```c
// All stride one access
// Significant data re-use
// Access array b in transpose form
for (i=0; i<n; i++) {
    for (i=0; i<n; i++) {
        for (k=0; k<n; k++) {
            c[i][j] += a[i][k] * b[j][k];
        }
    }
}
```
Code Examples (continued)

- Unavoidable bad stride
- Significant data re-use
- Matrix Multiple (C=AB)

```cpp
// Unavoidable bad stride access
// Significant data reuse
// c[i][j] has no stride (constant with respect to the inner loop
// a[i][k] is stride one
// b[k][j] is stride n
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        for (k=0; k<n; k++) {
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
    }
}
```

Basic Data Blocking

- Goal:
  - Restructure code such that array (matrix) access is such that once a block of data is allocated to the cache it is used as much as possible before it is discarded

![Diagram showing matrix multiplication](image)
Basic Data Blocking (continued)

• Goal:
  - Restructure code such that array (matrix) access is such that once a block of data is allocated to the cache it is used as much as possible before it is discarded.

Calculate a partial product using a block of data:

\[
\begin{array}{ccc}
  & c & \quad a & \quad b \\
\end{array}
\]

Data Blocked Example

```c
// a[i][j] is stride n
// b[j][i] is stride one
for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
        s=s+a[i][j]*b[j][i];
    }
}
```

```c
// NB is the blocking factor
for (jj=0; jj<n; jj+=NB) {
    for (ii=0; ii<n; ii+=NB) {
        for (j=jj; j<jj+NB; j++) {
            for (i=ii; i<ii+NB; i++) {
                s=s+a[i][j]*b[j][i];
            }
        }
    }
}
```