Performance Optimization

Dirk Pleiter

PDC and CST | EECS | KTH

January 2023

Overview

Computational Patterns

Memory Access Optimization

Cache Blocking

(Auto-)Vectorization

Content

Introduction to Computational Dwarfs

- **Dwarf** = An algorithmic method that captures a pattern of computation and communication
  - Terminology introduced in Krste Asanovic et al., “The Landscape of Parallel Computing Research: A View from Berkeley”, 2006
  - Various dwarfs already identified in earlier work
- Goal: Identify commonalities between numerical applications to exploit common knowledge and experience about
  - Performance characteristics (e.g. arithmetic intensity)
  - Best-practices for implementation on a given architecture

Computational Dwarfs: Overview

Dwarfs for HPC

1. Dense linear algebra
2. Sparse linear algebra
3. Spectral methods
4. N-body methods
5. Structured grids
6. Unstructured grids
7. MapReduce (Monte Carlo)

Other Dwarfs

8. Combinational logic
9. Graph traversal
10. Dynamic programming
11. Back-track and branch + bound
12. Graphical models
13. Finite state machines

Computational Dwarf: Structured Grids (1/3)

- Applications perform periodic updates of regular, multi-dimensional grids
- Memory access features
  - Regular strided memory access
  - High spatial data locality, i.e. consecutive access to data that is close in memory
- Application areas
  - Heat transfer
  - Lattice Quantum Chromodynamics
  - Computational fluid dynamics using the Lattice Boltzmann Method

Computational Dwarf: Structured Grids (2/3)

Example: 2-dimensional Poisson equation

- Formulation in the continuum

$$-\frac{\partial^2 v(x,y)}{\partial x^2} - \frac{\partial^2 v(x,y)}{\partial y^2} = f(x,y)$$

- Discretisation of 2nd-order derivative

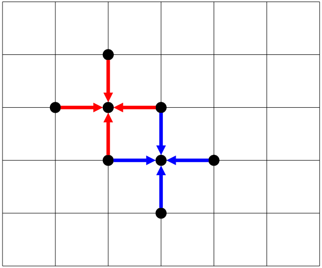
$$-\frac{\partial^2 v(x,y)}{\partial x^2} \leftarrow \frac{2v_{i,j} - v_{i-1,j} - v_{i+1,j}}{h^2}$$

- Discrete Poisson equaiton in 2 dimensions:

$$T\,v = h^2\,f \quad \text{where} \quad T = \begin{pmatrix} 4 & -1 & 0 & \cdots \\ -1 & 4 & -1 & \cdots \\ \vdots & \vdots & \ddots & \end{pmatrix}$$

Computational Dwarf: Structured Grids (3/3)

Graphical representation of $T\,v$:



Observations:

- Matrix T acts as a stencil operator
- Any element of vector v is reused 4 times ☞ data locality

Computational Dwarf: Dense Linear Algebra

$$\begin{pmatrix} 1 & 4 & 5 & 6 \\ 0 & 3 & 2 & 6 \\ 1 & 2 & 3 & 1 \\ 5 & 4 & 3 & 2 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 48 \\ 36 \\ 18 \\ 30 \end{pmatrix}$$

- Data are densely populated matrices or vectors
- Typical memory access pattern:
  Unit-stride or fixed-stride regular sequential access
- Example: $y_i \leftarrow \sum_j A_{ij} \cdot x_j$
- Popular library with dense linear algebra kernels: BLAS (Basic Linear Algebra Subprograms)
  - Different implementations available:
    - Open source: NETLIB BLAS, OpenBLAS, BLIS
    - Closed source: MKL (Intel), ESSL (IBM), Arm Performance Library (Arm)
  - Classification of operations
    - BLAS-1: scalar, vector and vector-vector
    - BLAS-2: matrix-vector
    - BLAS-3: matrix-matrix

Content

Digression: Memory Access Locality

- Empirical observation: Programs tend to reuse data and instructions they have used recently
- Observation can be exploited to
  - Improve performance
  - Optimize use of more/less expensive memory
- Types of localities
  - **Temporal locality**: Recently accessed items are likely to be accessed in the near future
  - **Spatial locality**: Items whose addresses are near one another tend to be referenced close together in time

Memory Access Locality: Example

```c
double a[N][N], b[N][N];

for (i=0; i<N; i++)
   for (j=1; j<N; j++)
      a[i][j] = b[j-1][0] + b[j][0];
```

- Assume right-most index being fastest running index
- Temporal locality: `b[j][0]`
- Spatial locality: `a[i][j]` and `a[i][j+1]` ($j + 1 < N$)

Memory Access Pattern Optimizations: Stride-1 Memory Access

```c
double a[N][N], b[N][N];

for (int j = 0; j < N; j++)
  for (int i = 0; i < N; i++)
    b[i][j] = a[i][j] + 0.1;
```

```c
double a[N][N], b[N][N];

for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    b[i][j] = a[i][j] + 0.1;
```

- Assume N to be very large
- Question: Which of the codes written in C will be faster? Why?

Stride-1 Memory Access (cont.)

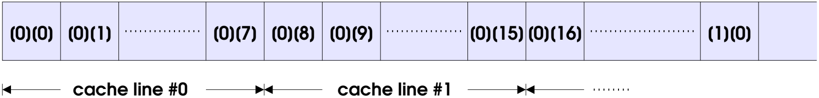
```c
double a[N][N], b[N][N];

for (int j = 0; j < N; j++)
  for (int i = 0; i < N; i++)
    b[i][j] = a[i][j] + 0.1;
```

```c
double a[N][N], b[N][N];

for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    b[i][j] = a[i][j] + 0.1;
```

- Answer: Right code because of stride-1 memory access
  - During each inner loop iteration of the left code 2 new cache lines are accessed
  - Because N is large, cache lines will be evicted before being reused
- The arrays are mapped to memory as follows assuming a cache line size of 64 Byte:



Memory Access Pattern Optimizations: Indirect Memory Access

Example: Sum over neighbours

```
double a[N], b[N];
int nnb[N];
int nb[N][NNB_MAX];

for (int i = 0; i < N; i++) {
  b[i] = 0;
  for (int j = 0; j < nnb[i]; j++)
    b[i] += a[nb[i][j]];
}
```

- Performance challenges
  - Memory latency becomes an issue as `a[]` can only be loaded after the load of `nb[][]` completed
  - Access to `a[]` likely not sequential with low stride
  - Vectorization of the code requires gather load instructions and the compiler auto-vectorizer may fail
- Possible optimizations strategies
  - Recompute index of neighbour
  - Improve spatial data locality by sorting data

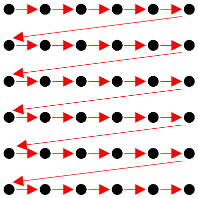
Content

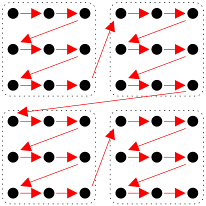
Cache Blocking

- **Cache blocking** is a strategy where data structures or memory access patterns are changed such that capacity cache misses are reduced
  - Optimisation strategy sometimes also called **loop tiling** or **loop blocking**
- Example: Double-precision matrix-vector multiplication (BLAS-2)

$$y_i \leftarrow \sum_{j=0}^{N-1} A_{ij}\, x_j \quad (i = 0, \ldots, N-1)$$

  - Advantage: x needs to be loaded only once per block
  - Disadvantage: Non-linear read of A





Cache-blocked Matrix-Vector Multiplication

```
double x[N], y[N];
double A[N][N];

for (int ib = 0; ib < N/B; ib++)
{
  for (int i = 0; i < B; i++)
    y[ib*B+i] = 0.0;

  for (int jb = 0; jb < N/B; jb++)
    for (int i = 0; i < B; i++)
      for (int j = 0; j < B; j++)
      {
        int ii = ib*B + i;
        int jj = jb*B + j;
        y[ii] += A[ii][jj] * x[jj];
      }
}
```

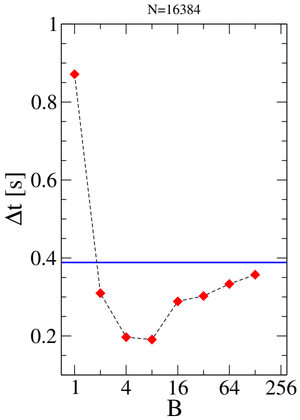
Cache-blocked Matrix-Vector Multiplication

- Let B be the block size with $B \leq N$ and N being a multiple of B
- Information exchange analysis:

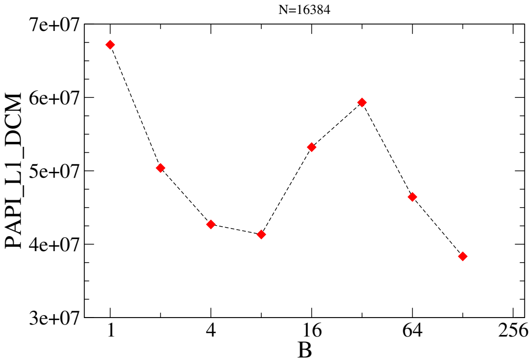
$$
\begin{aligned}
I_{\mathrm{rf,rf}}(N, B) &= [N + (N-1)]\; N\,\mathrm{Flop} \\
I_{\mathrm{mem,rf}}(N, B) &= (N + N^2 + N \cdot \mathbf{N/B})\,8\,\mathrm{Byte} \\
AI &= I_{\mathrm{rf,rf}} / I_{\mathrm{mem,rf}} \simeq \\
& (0.125\ \ldots\ 0.250)\,\mathrm{Flop/Byte}
\end{aligned}
$$

Cache-blocked Matrix-Vector Multiplication

- Results obtained for $N = 16384$ on Intel Xeon E5-2623 v3 using a single thread
  - Horizontal line shows results for no cache blocking
- Observations
  - Large number of nested loops generate a significant overhead
  - Up to $2\times$ speed-up compared to no cache blocking
  - Best results are obtained for small block sizes with $B = 8$



Cache-blocked Matrix-Vector Multiplication



- For small B number of L1 data cache misses drop as expected
- For large B number of L1 data cache misses drop due to memory pre-fetcher

Content

SIMD Parallelism

- Single Instruction Multiple Data (SIMD) instructions exploit data-level parallelism by operating on data items in parallel
  - E.g., SIMD add

$$\begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{pmatrix} \leftarrow \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

- Example ISA:
  - Intel Streaming SIMD Extensions (SSE)
  - Intel Advanced Vector Extensions (AVX, AVX2, AVX512)
  - POWER ISA (VMX/AltiVec, VSX)
  - Armv7 NEON, Armv8, Arm SVE

SIMD Programming: Auto-Vectorisation

Auto-vectorisation = Compiler capability to convert operations on scalars to operations on vectors

- Advantages:
  - Portable code
  - Programmer relieved from complex code transformations
- Disadvantage:
  - Only indirect control on code generation
  - Compiler may fail to identify vectorisation opportunities

Auto-Vectorisation: Example (1/3)

- daxpy (BLAS-1): $\vec{y} \leftarrow \alpha\,\vec{x} + \vec{y}$

```c
#define N 1024

void daxpy(double* x, double alpha, double* y)
{
  for (int i = 0; i < N; i++)
    y[i] += alpha * x[i];
}
```

- Using GCC compiler in a verbose mode:

```
% gcc -O3 -fopt-info -S daxpy1.c
...
daxpy1.c:6:21: optimized: loop vectorized using 16 byte vectors
daxpy1.c:6:21: optimized: loop versioned for vectorization because of possible
aliasing
daxpy1.c:3:6: note: vectorized 1 loops in function.
daxpy1.c:8:1: note: ***** Analysis failed with vector mode V2DF
daxpy1.c:8:1: note: ***** Skipping vector mode V16QI, which would repeat the
analysis for V2DF
```

- Compiler generates additional code to check possible aliasing issues at run-time

Auto-Vectorisation: Example (2/3)

- For `daxpy` we know that $\vec{x}$ and $\vec{y}$ are stored in different memory locations
- Use the `restrict` qualifier to inform the compiler about this:

```c
#define N 1024

void daxpy(double* restrict x, double alpha, double* restrict y)
{
  for (int i = 0; i < N; i++)
    y[i] += alpha * x[i];
}
```

- Recompiling modified code:

```
% gcc -O3 -fopt-info -S daxpy1.c
...
daxpy2.c:6:21: optimized: loop vectorized using 16 byte vectors
daxpy2.c:3:6: note: vectorized 1 loops in function.
daxpy2.c:8:1: note: ***** Analysis failed with vector mode VOID
```

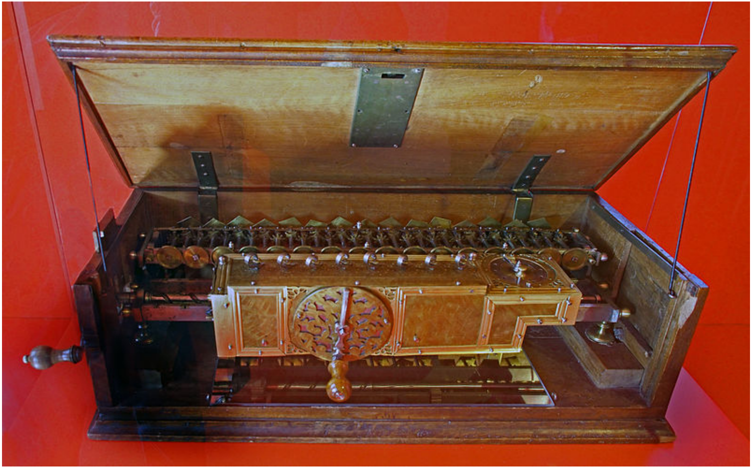
Auto-Vectorisation: Example (3/3)

- Assembler generated for the Kunpeng 920 processor:

```
daxpy:
.LFB0:
        .cfi_startproc
        dup     v2.2d, v0.d[0]
        mov     x2, 0
        .p2align 3,,7
.L2:
        ldr     q0, [x1, x2]
        ldr     q1, [x0, x2]
        fmla    v0.2d, v2.2d, v1.2d
        str     q0, [x1, x2]
        add     x2, x2, 16
        cmp     x2, 8192
        bne     .L2
        ret
        .cfi_endproc
```

- Advice for reading the assembler:
  - Code uses 128-bit SIMD registers (suffix `.2d`)
  - Address pointer stored in register `x2` is incremented by 16 Byte in each loop iteration

Finish with a Simple Architecture: Leibniz' Reckoner



[Museum Schloss Herrenhausen]