SE205 - TD1 Énoncé

General Instructions

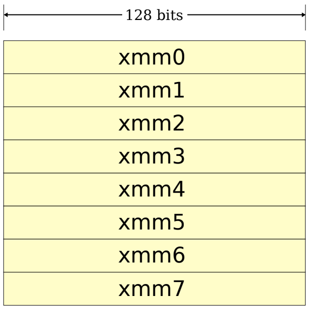
- You can download all source files from the assignment on Moodle.

SIMD-like Data-Level Parallelism

Modern processors often come with instruction set extensions that allow to exploit data-level parallelism. These extensions are often called multimedia extensions or vector extensions, and basically consist of vector registers and SIMD-like instructions operating on these vector registers. Intel processors for instance offer several such extensions ranging from MultiMedia eXtensions (MMX), over Streaming SIMD Extensions (SSE), to the new Advanced Vector Extensions (AVX). Similar extensions are available for many other architectures, e.g. on ARM VFP/NEON, on PowerPC AltiVec, on SPARC VIS, on MIPS MDMX/MIPS-3D/MSA.

For the exercises we will rely on Intel's (pretty old) SSE extension for illustration. SSE provides 8 additional 128-bit registers (`%xmm0, ..., %xmm7`). These registers can hold various forms of vectors, ranging from 16-element vectors, where each element has 8 bits, up to 2-element vectors, with elements of 64 bits each. The registers can also be used to hold 128-bit scalar values. The 128-bit values stored in the registers do not give any indication about the actual vector format used. This is specified by the instructions operating on the registers.

Figure 1: 128-bit registers introduced by SSE



SSE also introduced vector instructions (in SSE-terminology *packed*) as well as related instructions to create/manipulate vectors and their elements. The most important instructions (that we will see later on) are described here (attention this is GNU-style assembly, i.e., the operand order does not match that of the Intel manuals):

movdqu (mem)|%xmm$s$, %xmm$d$

Move a double quadword from a memory location `mem` or vector register `%xmm`s to vector register `%xmm`d. The memory address may be unaligned.

movdqu %xmm$s$, (mem)
: Move a double quadword from the vector register %xmm$s$ to memory location mem. The memory address may be unaligned.

movdqa %xmm$s$, (mem) and **movdqa (mem)|%xmm$s$, %xmm$d$**
: Same as movdqu. Memory addresses have to be aligned with the vector size.

paddd (mem)|%xmm$s$, %xmm$d$
: Read a vector from a memory address or the %xmm$s$ register and perform a vector addition with the vector in %xmm$d$. The result is stored in %xmm$d$. The vectors consist of 4 elements, 32 bits each.

pmuludq (mem)|%xmm$s$, %xmm$d$
: Read a vector from a memory address or the %xmm$s$ register and perform a vector multiplication with the vector in %xmm$d$, considering the *even* 32-bit elements only. The 64-bit results are stored in the vector %xmm$d$. The input vectors consist of 4 elements, 32 bits each, of which only two are used. The result is a vector with two 64-bit elements.

pshufd imm, (mem)|%xmm$s$, %xmm$d$
: Read a vector from a memory address or the %xmm$s$ register and shuffle its elements based on the immediate operand imm. The result is stored in %xmm$d$. The vectors consist of 4 elements, 32 bits each.

punpckldq (mem)|%xmm$s$, %xmm$d$
: Read a vector from a memory address or the %xmm$s$ register and interleave the two lower elements of %xmm$s$ with the lower two elements of %xmm$d$. The result is stored in %xmm$d$. The vectors consist of 4 elements, 32 bits each.

psrlq imm, %xmm$d$
: Read a vector from the %xmm$d$ register and shift each of its elements to the right based on the immediate operand imm, while shifting 0 bits in. The result is stored in %xmm$d$. The vector consists of 2 elements, 64 bits each.

You can find more information about SSE on Wikipedia and the website of Intel.

Using the cachegrind Profiler

Valgrind is a very popular tool to find memory leaks in C and C++ programs – just in case you did not know. Apart from the memory checker, Valgrind also proposes additional tools too profile programs. One such tool is called Cachegrind. Cachegrind allows you to run a program and collect statistics about a (simulated) cache and its behavior while executing the program.

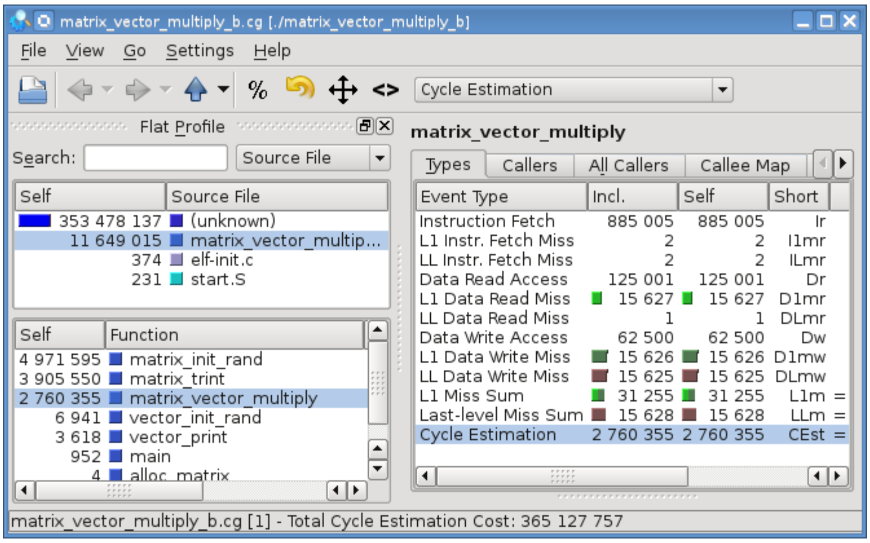
The Cachgrind tool can simply be invoked using the following command line:

```
valgrind --cachegrind-out-file=<your-log-file> --tool=cachegrind <your-program>
```

This will generate a profile with regard to the memory accesses and the cache behavior of the given program and store it in the file indicated by the first option (--cachegrind-out-file).

The trace can then be analyzed using the tool kcachegrind by invoking:

Figure 2: The graphical interface of `kcachegrind`.



Self	Source File
353 478 137	(unknown)
11 649 015	matrix_vector_multip...
374	elf-init.c
231	start.S

Self	Function
4 971 595	matrix_init_rand
3 905 550	matrix_trint
2 760 355	matrix_vector_multiply
6 941	vector_init_rand
3 618	vector_print
952	main
4	alloc_matrix

Event Type	Incl.	Self	Short
Instruction Fetch	885 005	885 005	Ir
L1 Instr. Fetch Miss	2	2	I1mr
LL Instr. Fetch Miss	2	2	ILmr
Data Read Access	125 001	125 001	Dr
L1 Data Read Miss	15 627	15 627	D1mr
LL Data Read Miss	1	1	DLmr
Data Write Access	62 500	62 500	Dw
L1 Data Write Miss	15 626	15 626	D1mw
LL Data Write Miss	15 625	15 625	DLmw
L1 Miss Sum	31 255	31 255	L1m =
Last-level Miss Sum	15 628	15 628	LLm =
Cycle Estimation	2 760 355	2 760 355	CEst =

matrix_vector_multiply_b.cg [1] - Total Cycle Estimation Cost: 365 127 757

```
kcachegrind <your-log-file>
```

The information is available for individual functions, which can be selected on the left hand side. On the right the various forms of cache accesses are shown. For this TD we will mostly focus on the following fields in the column “Self”, i.e., the third column:

Instruction Fetch The number of instructions executed (lower is better).

Data Read Access The number of data accesses (memory loads) executed (lower is better).

L1 Data Read Miss The number of data accesses that caused a cache miss (lower is better).

Using the Debugger GDB

We will have a look at assembly code in this TD and, probably, will also want to see what that code is doing. The best way to do this (on Linux) is to use the debugger GDB. Here is a quick guide, covering only commands needed for this TD. Instead of using GDB from the command line we will use the graphical front end `kdbg`. The debugger is invoked using:

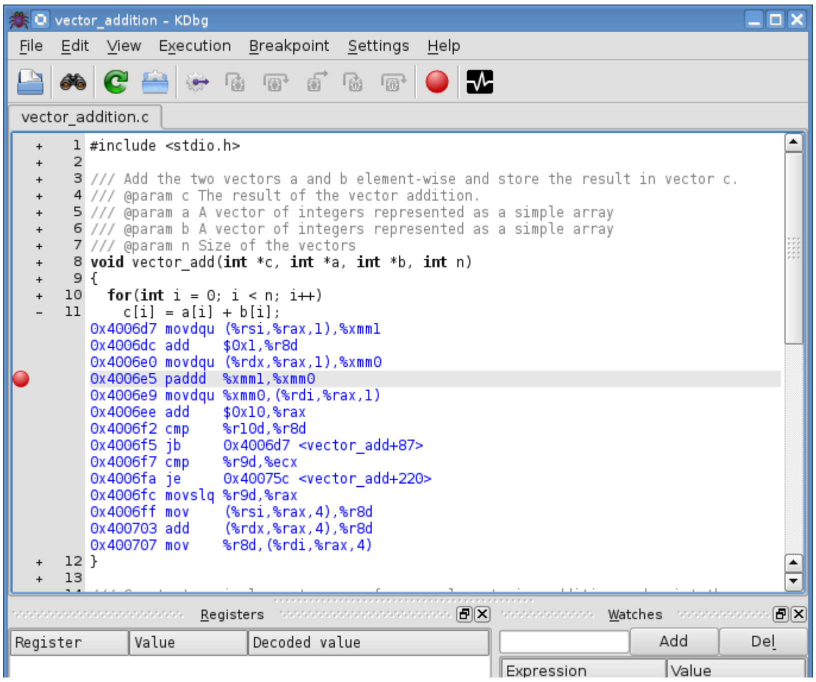
```
kdbg <your-program>
```

You can find kdbg at: `/cal/homes/brandner/opt/kdbg-2.5.5/bin/kdbg`

The window then shows the program’s source code, the status of the program, including values of variables, the call stack, as well as the values of the processor registers.

The source code view allows you to have a look at the corresponding *assembly code*, by clicking on the + symbol on the left hand side.

Figure 3: The graphical interface of `kdbg`.



```
#include <stdio.h>

/// Add the two vectors a and b element-wise and store the result in vector c.
/// @param c The result of the vector addition.
/// @param a A vector of integers represented as a simple array
/// @param b A vector of integers represented as a simple array
/// @param n Size of the vectors
void vector_add(int *c, int *a, int *b, int n)
{
  for(int i = 0; i < n; i++)
    c[i] = a[i] + b[i];
0x4006d7 movdqu (%rsi,%rax,1),%xmm1
0x4006dc add    $0x1,%r8d
0x4006e0 movdqu (%rdx,%rax,1),%xmm0
0x4006e5 paddd  %xmm1,%xmm0
0x4006e9 movdqu %xmm0,(%rdi,%rax,1)
0x4006ee add    $0x10,%rax
0x4006f2 cmp    %r10d,%r8d
0x4006f5 jb     0x4006d7 <vector_add+87>
0x4006f7 cmp    %r9d,%ecx
0x4006fa je     0x40075c <vector_add+220>
0x4006fc movslq %r9d,%rax
0x4006ff mov    (%rsi,%rax,4),%r8d
0x400703 add    (%rdx,%rax,4),%r8d
0x400707 mov    %r8d,(%rdi,%rax,4)
}
```

You can set a *break point* using the red button at the top of the window. Simply select a line or assembly instruction in the source code view and click on the button. The debugger will stop the program at this point whenever the execution reaches the break point.

You can also look at certain expressions that might be interesting to follow during the program's execution in the *watch* window. Simply type an arbitrary expression into the command line and click on the `add` button. For the TD we will want to watch the contents of the SSE vector registers, and here in particular we want to see them as 4-element vectors of 32-bit integers. This can be done using the following *watch* expression: "$xmm0.v4_int32" (since the value is a vector you need to unfold it to see the element values, also note the $ symbol).

1 Vector Reduction (30 minutes)

Aims: *Reason about dependencies in programs with loops.*

- Draw a diagram representing the data dependencies of the function `vector_reduction_sum` (see below). You can focus on accesses to the variable `sum` and the array `a`.

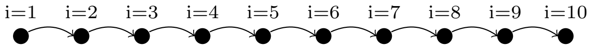
  Hint: Pay special attention to the `+=` operator.

  There is a loop-carried dependence between subsequent iterations on the variable `sum`. Since array elements are only read, only a true-dependencies with respect to `a` exist.

- Are there any loop-carried dependencies?

Yes, for variable `sum`; the distance is +1.

- How does the iteration space graph look like?



- What can you say about the potential for parallelization using SIMD-like vector operations?

  SIMD-like parallelization is not immediately possible due to the loop-carried dependencies.

```
#include <stdio.h>

/// @param a A vector of integers represented as a simple array
/// @param n Size of the vector
/// @return The sum over all elements in the vector, i.e.,
/// a[1] + a[2] + ... + a[n-1]
int vector_reduction_sum(int *a, int n)
{
  int sum = 0;
  for(int i = 0; i < n; i++)
    sum += a[i];

  return sum;
}

/// Compute the sum over all elements of a simple vector and terminate.
/// @return Always returns 0.
int main()
{
  // initialize a simple vector
  int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  // compute and print the sum over all vector elements
  int sum = vector_reduction_sum(a, sizeof(a) / sizeof(int));
  printf("Sum:␣%d\n", sum);

  return 0;
}
```

2 Vector Addition (30 minutes)

Aims: *Reason about dependencies and relate them to compiler optimizations.*

- Without knowing the contents of function `main`, what can you say about the pointers to the arrays `a`, `b`, and `c` passed to `vector_add` (see below)?

  Nothing, they might be identical, might partially overlap, or might be independent.

- Draw a diagram representing the data dependencies of the function `vector_add`. You can focus on accesses to the arrays `a`, `b`, and `c`.

  Since nothing can be said about the relation between pointers `a`, `b`, and `c` one has

to assume that loop-carried dependencies among the write accesses to `c` and the read accesses to `a` and `b` exist.

- Compile the program using the following command:

  ```
  gcc -fopt-info-vec-optimized -std=c99 -fno-inline \
      -msse -O3 -g -o vector_addition td1.2_vector_addition.c
  ```

- Inspect the generated assembly code using the following command

  ```
  objdump -dS vector_addition
  ```

  Then, verify what the code is doing at runtime using `kdbg`.

  Hint: Look at the assembly code of the addition in the loop. Try to find SSE operations in the assembly code and use a break point to see whether these instructions are executed. Try to modify the code and see whether changing the parameters to the function changes the behavior, e.g., replace parameter `dest` by `src1` in `main`, ...

- Look at the output of the compiler and the generated assembly code. What did the compiler do? Did the compiler succeed to vectorize the code? Are all elements added using vector instructions? It seems that much more code than a simple loop and an addition is executed, what might be the reason for this?

  Hint: Relate the assembly code to the answers you gave to the questions from above.

  The compiler performs tests on the pointers to ensure that they are aligned, do not overlap, and that `n` is larger than the vector size (4 in the case of 32-bit integers with Intel's SEE extension). If these tests succeed it simply performs vector operations, otherwise it falls back to a simple non-parallel loop.

```
#include <stdio.h>

/// Add the two vectors a and b element-wise and store the result in vector c.
/// @param c The result of the vector addition.
/// @param a A vector of integers represented as a simple array
/// @param b A vector of integers represented as a simple array
/// @param n Size of the vectors
void vector_add(int *c, int *a, int *b, int n)
{
  for(int i = 0; i < n; i++)
    c[i] = a[i] + b[i];
}

/// Print the value of each element of a vector.
/// @param v The vector to print.
void vector_print(int *v, int n)
{
  printf("(");
  for(int i = 0; i < n; i++)
  {
    if (i == n - 1)
      printf("%d)\n", v[i]);
    else
      printf("%d␣", v[i]);
  }
}

/// Create two simple vectors, perform an element-wise addition, and print the
```

```
/// result.
/// @return Always returns 0.
int main()
{
  // initialize two simple vector
  int src1[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int src2[] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

  // the size of the vectors
  int vsize = sizeof(src1) / sizeof(int);

  // allocate space for an additional vector
  int dest[vsize];


  // perform the vector addition
  vector_add(dest, src1, src2, vsize);

  // print the vectors
  printf("src1:␣"); vector_print(src1, vsize);
  printf("src2:␣"); vector_print(src2, vsize);
  printf("dest:␣"); vector_print(dest, vsize);

  return 0;
}
```

3 Matrix Vector Product (30 minutes)

Aims: *Reason about memory accesses in loops, vectorization, and cache performance.*

- Compile the program using the following command:

  ```
  gcc -fopt-info-vec-optimized -std=c99 -fno-inline \
      -msse -O3 -g -o matrix_vector_multiply td1.3_matrix_vector_multiply.c
  ```

- Recompile the program using the following command:

  ```
  gcc -fopt-info-vec-missed -std=c99 -fno-inline \
      -msse -O3 -g -o matrix_vector_multiply td1.3_matrix_vector_multiply.c
  ```

- Check the compiler messages. The compiler apparently was not able to vectorize the code. Can you give an explanation?

  Hint: Think about the dependencies, the iteration space graph, and the way the elements in the matrix are accessed in the loop of the function `matrix_vector_multiply` (no need to draw any diagram, though).

  The accesses in the inner loop traverse the matrix column-wise. The compiler thus cannot immediately form vectors. This would only work with the elements in a row.

- Make a copy of the original source file and try to change the source code of the function such that the compiler can vectorize it. Explain why your modification resolves the problem from above.

  The two loops have to be interchanged, i.e., the outer loop has to use the counter

variable `j` and the inner variable `i`. The inner loop now visits the matrix elements row-wise, which allows their elements to be packed immediately into vectors.

- Compile both programs and run them using `cachegrind`, as shown be the following example:

  ```
  valgrind --cachegrind-out-file=matrix_vector_multiply.cg \
            --tool=cachegrind ./matrix_vector_multiply \
            > matrix_vector_multiply.txt
  ```

- What do you notice with regard to the number of instructions executed by your two programs (just look at the function `matrix_vector_multiply` and ignore the other functions)? What can you say about the data cache behavior?

  The vectorized version executes fewer instructions and causes considerably fewer data cache misses.

- Where is most of the speed-up coming from? Is it really due to the vectorization?

  In comparison to the original program, most of the speed-up is due to the reduction in cache misses (the speed-up is about 2.3x). The vectorization then still gives considerable improvements of 22%, however, this is small in comparison to the reduction due to the cache misses.

```c
#include <stdio.h>
#include <stdlib.h>

/// define the size of the matrices
#define SIZE 500

/// Matrix representation as two-dimensional array
typedef int (*matrix_t)[SIZE];

/// Vector representation as a simple array
typedef int vector_t[SIZE];

/// Multiply matrix m with vector v and store the result in d.
/// @param d The result matrix.
/// @param m The input matrix.
/// @param v The input vector.
void matrix_vector_multiply(matrix_t d, matrix_t m, vector_t v)
{
  for(int i = 0; i < SIZE; i++ )
  {
    for(int j = 0; j < SIZE; j++ )
    {
      d[j][i] = m[j][i] * v[i];
    }
  }
}

/// Initialize a given matrix with random values (module 5).
/// @param a The matrix to initialize.
void matrix_init_rand(matrix_t a)
{
  for(int i = 0; i < SIZE; i++ )
  {
    for(int j = 0; j < SIZE; j++ )
```

```
    {
      a[i][j] = rand() % 5;
    }
  }
}

/// Print the elements of a matrix.
/// @param m The matrix to print.
void matrix_print(matrix_t m)
{
  for(int i = 0; i < SIZE; i++ )
  {
    for(int j = 0; j < SIZE; j++ )
    {
      printf("%3d␣", m[i][j]);
    }
    printf("\n");
  }
}

/// Allocate a matrix of SIZE x SIZE elements on the heap.
/// @return A pointer to a newly allocated block of memory for a matrix.
matrix_t matrix_alloc()
{
  void *p = malloc(sizeof(int) * SIZE * SIZE);
  return p;
}

/// Initialize a given vector with random values (module 5).
/// @param v The vector to initialize.
void vector_init_rand(vector_t v)
{
  for(int i = 0; i < SIZE; i++ )
  {
    v[i] = rand() % 5;
  }
}

/// Print the value of each element of a vector.
/// @param v The vector to print.
void vector_print(int *v)
{
  printf("(");
  for(int i = 0; i < SIZE; i++)
  {
    if (i == SIZE - 1)
      printf("%d)\n", v[i]);
    else
      printf("%d␣", v[i]);
  }
}

/// Create an input matrix and vector with random data, multiply them, and
/// display the result.
/// @return Always returns 0.
int main()
{
  // allocate two matrices and a vector
  matrix_t dest = matrix_alloc();
  matrix_t msrc = matrix_alloc();
  vector_t vsrc = {0,};

  // initialize two of them
  matrix_init_rand(msrc);
  vector_init_rand(vsrc);

  // perform the matrix vector product
  matrix_vector_multiply(dest, msrc, vsrc);

  // print the matrices and the vector
  printf("vsrc:\n"); vector_print(vsrc);
  printf("msrc:\n"); matrix_print(msrc);
  printf("dest:\n"); matrix_print(dest);

  // free dynamically allocated memory
  free(dest);
  free(msrc);

  return 0;
}
```