TÉCNICO LISBOA

Assignment 1B

- Tutorial -

Parallel and Heterogeneous Computing Systems

Computação em Sistemas Paralelos e Heterogéneos (CSPH)

2024/2025

Overview

This assignment is intended to help you develop an understanding on the following topics:

1. How to program using intrinsics to take advantage of the performance benefits of SIMD.
2. Learn how to speed up scalar code via the use of intrinsics and understand the different aspects of good SIMD performance and how to achieve them.

Deadlines and Submission

NO SUBMISSIONS! This work serves as a tutorial and preparation – **NO EVALUATION**. It is intended for your practice and learning via examples and tutorials. This assignment provides a set of exercises for you to practice! You can resolve them at our lab session or at home. Surely, you can discuss your solutions and doubts with the teacher of your labs in our next lab session. Again, there is no submission for this part of the work and your solutions will not be graded!

You can consider these examples as a representative set of problems to get you prepped for the work to be developed in the class (which will be evaluated). Those assignments are not yet provided to you!

Environment Setup

To get started:

1. The assignment starter code is available on the course webpage (section: Labs). `wget` it, `unzip` it, and start having fun with it.

The assignment starter code is provided in the *fake_intrinsics* folder. It is composed of two header files (`fake_intrinsics.h` and `stats.h`), the main file (`main.cpp`), and the `Makefile`. To compile the code, all that is necessary to do is to run the command `make` in a terminal in the same directory as the code and then to run it just use ./my_program

The file `stats.h` holds a structure to measure the following metrics from your SIMD program: *i)* the number of available SIMD lanes, *ii)* the total number of program cycles, *iii)* the total number of vector instructions, *iv)* the maximum number of active lanes during the program execution, *v)* the actual number of active lanes during the program execution, and *vi)* the vector utilization. One instance of this structure is present in `fake_intrinsics.h` to count statistics.

The file `fake_intrinsics.h` contains all the fake intrinsics operations covered in the theoretical classes (see Lecture 2). In this file, you can define the SIMD length (`VECTOR_LENGTH`) and the array size to process (`N`). Whenever you run an operation, the name of the operation is printed on the terminal, as well as the SIMD lanes that are active during that operation. To visualize intermediate results between computations, `fake_intrinsics.h` also provides a function to print vectors (`printVector`). After all the computations are done, using the function `printStats` will print on the terminal the stats of the program. Regarding instructions, the basic types (`__vint`, `__vfloat`, and `__vbool`, which are integers, floating-point numbers, and booleans, respectively) are allowed in all instructions, except in `__vload` and `__vstore` instructions, where the type `__vbool` is **not allowed**.

Problem 1: Absolute value of an array (problem from the theoretical lecture)

Take a look at the function from the theoretical class (Lecture 2) that computes the absolute value of each element, implemented in scalar code and with our fake intrinsics. Rather than craft an implementation using SSE or AVX2 vector intrinsics that map to real SIMD vector instructions on modern CPUs, to make things a little easier, we have implemented a version of this code with our "fake processor" using our "fake vector intrinsics". To be precise, in the `main.cpp` file provided in the `fake_intrinsics` folder, you can find this SIMD program using the fake intrinsics, which in conjunction with `fake_intrinsics.h` to define the SIMD length (`VECTOR_LENGTH`) and the array size to process (`N`) should allow you to "scale" your code across "fake processors" with different SIMD widths.

Our simple code: absolute value of each element

```
float* in = new float[N];
float* out = new float[N];

// initialize "in" here

for (int i=0; i<N; i++)
{
   float x = in[i];
   float y;

   if (x < 0)
     y = -x;
   else
     y = x;

   out[i] = y;
}
```

Our simple code: with "fake" intrinsics

```
float* in = new float[N];
float* out = new float[N];

// initialize "in" here

for (int i=0; i<N; i+=8)
{
   __vfloat x = _vload(&in[i]);
   __vfloat y;
   __vfloat zeros = _vbcast(0.f);
   __vbool mask = _vlt(x,zeros);
   y = _vsub(zeros,x,mask);
   mask = _vnot(mask);
   y = _vcopy(y,x,mask);

   _vstore(&out[i],y);
}
```

What you need to do:

1. Can you roughly match the lines of code in the scalar code with the corresponding ones in the fake intrinsics (hint: some scalar instructions might map to more than just one vector instruction)?
2. Can you comment on each line of the vector code describing what it's doing?
3. Run the code and try different SIMD `VECTOR_LENGTH` values (2, 4 and 8). What do you observe in terms of the total number of cycles, vector instructions and utilization? Can you explain it?

Problem 2: Vectorizing Code using our "fake" SIMD Intrinsics

Take a look at the function `clampedExpSerial` provided below. The `clampedExp()` function raises `values[i]` to the power given by `exponents[i]` for all elements of the input array and clamps the resulting values at 9.999999. In Problem 2, your job is to vectorize this piece of code, so it can "run" on a machine with SIMD vector instructions.

Here are a few hints to help you in your implementation:

- Consider that the total number of loop iterations is always a multiple of SIMD vector width.
- *Hint:* For this exercise `_vpopcnt` (popcount) intrinsic is introduced in our "fake vector intrinsics". Popcount instructions count the number of set bits in a value, i.e., it returns a scalar value (integer). Using the `_vpopcnt` intrinsic on `__vbool` data may be helpful in this problem

```
// For each element, compute values[i]^exponents[i] and clamp value to 9.999.
// Store result in output.
void clampedExpSerial(float* values, int* exponents, float* output, int N) {
  for (int i=0; i<N; i++) {
    float x = values[i];
    int y = exponents[i];
    if (y == 0) {
      output[i] = 1.f;
    } else {
      float result = x;
      int count = y - 1;
      while (count > 0) {
        result *= x;
        count--;
      }
      if (result > 9.999999f) {
        result = 9.999999f;
      }
      output[i] = result;
    }
  }
}
```

For the execution, consider the following input values:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
values	3.0	2.0	2.5	1.25	5.5	0.5	10.1	3.15	1.75	6.55	1.63	1.5	4.33	0.15	1.95	2.83
exps.	0	2	3	10	0	4	0	3	5	1	4	9	0	5	0	3

You should consider the performance of your implementation to be the value "Total Vector Instructions" and “Execution Time [Cycles]”. (Our C++ simulator assumes that every “fake” instruction and main loop iteration take one cycle on the fake SIMD CPU). It also calculates the "Vector Utilization", i.e., the percentage of vector lanes that are enabled during the execution of your code.

What you need to do:

1. Consider a CPU with the vector width of 8. Implement a vectorized version `clampedExpVector` of `clampedExpSerial` in the main.cpp file. The correct solution to this problem should aim to minimize the amount of scalar instructions besides the inner while control. Report the "Total Vector Instructions" and "Vector Utilization" metrics. Don't forget to initialize the arrays `values` and `exponents` with the values presented above.

2. Consider that we change the vector width from 4, 8 to 16 (change the value of VECTOR_LENGTH)). Does the vector utilization increase, decrease or stay the same as the vector width changes? Why? What about speedup (execution time)?