

Computer Architecture Project

Computer Architecture and Technology Area – Version 1.1.12225, 02/10/2024

Table of Contents

Objectives

1. Development environment
2. Project development
 - 2.1. Single-threaded program (reference, 2 points)
 - 2.2. Phase 2 of the teamwork
3. Milestone

Appendix A: Test images

Appendix B: Visual Studio Code introduction

Appendix C: Assessment matrix

Appendix D: SIMD references

Objectives

- Illustrating performance improvements that can be achieved in a real computer by using multi-threaded programming to take advantage of concurrent execution.
- Programming SIMD instructions provided by some architectures and measuring the performance improvement.
- Developing teamwork and documentation skills, which are essential for a computer engineering.

1. Development environment

A virtual machine with all the software needed is provided for developing the project. This virtual machine is configured with a predefined user and password, which are `student` and `teamwork` respectively. This machine includes the following software:

- Ubuntu Desktop 18.04.3 LTS with minimal installation and the following packages: *build-essential*, *git*, *cimg-dev* and *openssh-server*.
- Visual Studio Code with the following extensions: *C/C++* and *TODO Highlight*.

The network of the virtual machine is configured as NAT and a port forwarding rule is included in case you need to connect using SSH. You must use address `localhost` and port `61022` to connect to the virtual machine.



- Download the [file with the virtual machine](https://www.atc.uniovi.es/grado/2ac/2021-teamwork.ova) (<https://www.atc.uniovi.es/grado/2ac/2021-teamwork.ova>) to your computer.



This virtual machine is different from the one you have been using in the labs.

- Open VirtualBox, go the menu option **File › Import Appliance** and choose the file that you have already downloaded.
- Use the default importing options and press [**Import**].
- Before starting the virtual machine configure it to use the same number of CPU cores as the physical machine (at least two are necessary).
- Take a screenshot of the **Settings › Processor** tab of the virtual machine in VirtualBox. This screenshot must be included in the documentation.
- Configure the memory of the virtual machine with a minimum of 3 GiB, unless your machine has more than 6 GiB installed. In this case you should configure it with half the available memory.
- Start your virtual machine.
- Go to menu option **Devices › Insert Guest Additions CD image** of VirtualBox and press [**Run**] when prompted. You may need to provide your user password.
- If at any time you are requested to install updates, do so. You will see an icon in the menu bar showing the progress of the update.
- Restart your virtual machine.
- Save a copy of the `/proc/cpuinfo` file of the virtual machine. This should be also included in the documentation.

2. Project development

During the project the team should develop three C/C++ programs and perform several performance and acceleration measurements for each of them. In addition, a report must be written describing the steps followed to achieve the proposed solutions.

2.1. Single-threaded program (reference, 2 points)

Each team should write a program in C/C++ to implement operations on digital images (digital filters). The characteristics of the images and the type of operations will be provided by the teacher.

The program to write consists on a basic implementation and will be taken as a reference to calculate the speedup of subsequent implementations.

All filter implementations, regardless of the operations to be performed, must fit the following prototype:

```

1  typedef struct {
2      data_t *pRsrc;
3      data_t *pGsrc;
4      data_t *pBsrc;
5      data_t *pRdst;
6      data_t *pGdst;
7      data_t *pBdst;
8      uint pixelCount;
9  } filter_args_t;
10
11 void filter(filter_args_t pArgs);

```

In the structure shown, the data in code lines from 2 to 4 are pointers to the memory areas where the RGB components of the (source) image to be filtered are located. The data in code lines from 5 to 7 are pointers to the memory area where the components of the (destination) image, that results after the applied filter, are located. The integer `pixelCount` is the size of the image in pixels.

A C++ project for Visual Studio Code is provided in a Bitbucket git repository. You can find a brief introduction to Visual Studio Code in Visual Studio Code introduction.

ONE AND ONLY ONE of the members of the team must fork this repository into a private repository in his/her Bitbucket account following the steps listed below:



- Log in into [Bitbucket](https://bitbucket.org) (<https://bitbucket.org>) using a new tab of the web browser.
- Go to the [public repository](https://bitbucket.org/2acuniovi/2023-single-thread/src/master/) (<https://bitbucket.org/2acuniovi/2023-single-thread/src/master/>) of the project.
- Click on the **+** symbol in the menu on the left and choose **Fork this repository**.
- Choose the default name for the repository adding the team name at the end. For instance, if the team name is **PL4-B**, the name of the repository should be **2023-single-thread-pl4-b**. **You must enable the checkbox that makes the repository private**, otherwise it will be public.
- Create a Group in the Workspace and add your team mates and your professor to that group. you can do this from **Settings › Workspace Settings › User Groups** (If the BitBucket account is new, the interface may vary slightly. You will need to go to **Settings › Workspace Settings › Workspace Permissions › product access**).
- Grant write access to the group you just created to the repository by clicking **Invite › Repository permissions › Add users or groups**.

Once these steps are completed, every member of the team must clone this new repository. This can be done from the command-line interface as explained in lab session 0.



- Configure your git username and email from a command-line interface as described in lab session 0.

```

$> git config --global user.name "<user>"
$> git config --global user.email "<email>"


```

As you did in the lab session 0, if you want to access the remote repository in Bitbucket from the command interface it is necessary to create what is known in Bitbucket terminology as an *app password*.

- Click on the gear that is on the upper right corner of the Bitbucket page. Select **Personal BitBucket settings** and, in the left menu, select **App passwords**. Then press the [**Create app password**] button. You must provide a label of your choice to distinguish it from other future app passwords. In addition, you must enable at least all permissions in the **Repositories** section.
- Press the [**Create**] button to create the app password.
- Copy and paste the app password in a text document and keep it in a safe place as you will need it soon.
- Navigate back on the Bitbucket page to your repository and click the [**Clone**] button. Copy the text that appears in the popup window (it's a command).
- Go to your home directory in the Linux command-line interface and execute the command you have already copied in order to clone the Bitbucket repository to your machine.
- Once requested, enter the *app password* just copied into the text file. Remember that you can paste this password into the **putty** window by clicking the right mouse button. Although apparently nothing changes, the password is pasted. It is simply hidden for security reasons. Press the key so the repository is cloned locally.

A more convenient approach is working with git addon for Visual Studio Code. In this case, you may follow these steps:



- Open Visual Studio Code IDE by clicking in the  icon placed on the side bar of the desktop.
- Choose menu option **View > Command Palette**. A text box is open on top of the IDE window.
- Type **git clone** in this text box and press .
- Type the URL of the **private repository** and press .



You can copy and paste the URL if you have the clipboard sharing enabled for the virtual machine.

- Select the directory where to clone the repository using the dialog that appears.
- Type your Bitbucket username and password in the aforementioned text box.
- Choose [**Open**] when prompted on the bottom right of the IDE to open the downloaded git repository.

From now on, you can work with the repository from the command-line interface or use the integrated version control of Visual Studio Code. The use of the repository for coordinating the work is highly recommended.



Various team members may work on the same source code file simultaneously committing changes to the repository without creating conflicts provided they work on different lines. If another member has already uploaded commits on the file to the remote repository when you *git push*, you will have to pull remote commits before pushing yours. A commit to merge your commits with those downloaded is automatically triggered. You may simply accept the default commit message.

The project in the repository is used to build **single-thread**. This program is similar to the one developed in lab session 1.2. However, the operations the algorithm must perform are different. The program measures the time spent to execute the operations on the images, without including the time required to load, initialize and save images to disk.



The **Makefile** of the project includes **debug** and **release** building rules, although only **debug** is available from Visual Studio Code.

The project can be built and executed, but you need to copy some test image into the root folder of the project. When building within Visual Studio Code, the project is built with debugging options enable.



By default, the program expects a file called **bailarina.bmp** to be located in the root directory of the project.

You can download a set of test images following the steps described in Test images.

Now, you are ready to modify the program to implement your algorithm.



- Edit and complete the `main.cpp` file to deal with all **TODO** and **FIXME** labels in the code. You must implement the image operation with the component data type you were assigned.
- Copy to the root directory of the project the images from the **Pictures** directory that the program is going to process.
- Build and debug the program from Visual Studio Code as described in Visual Studio Code introduction as many times as necessary until it is correct.
- Run the **Release** version of the program from the command-line interface once to check how much time it needs to execute. You can build the release version of the program by issuing this command in the root directory of the project:

```
$> make release
```



The release version is built in the **release** directory. Bear in mind that the images to process are located in the project root directory. Thus, you must move the command-line interface to this directory and execute the program as follows.

```
$> release/single-thread
```

- If the time needed is not between 5 and 10 seconds, modify the source code to repeat the algorithm in a loop as many time as necessary for the elapsed time to be in this interval.



This is to obtain significant speedups when comparing with the programs that are going to be written in phase 2.

- Once the elapsed time is between 5 and 10 seconds, execute the program 10 times in **Release** mode using the command-line interface.



You must close Visual Studio Code and any other program to carry out the execution, so the elapsed time is not affected by other running programs.

- Compute the average, the standard deviation and the confidence interval of the mean (95%) of the elapsed time.
- All measurements and statistics should be written down in a table to be included in the documentation.

2.2. Phase 2 of the teamwork

Two new programs are written in this phase. They can be built using two Visual Studio Code projects located in two git public repositories: [SIMD version](https://bitbucket.org/2acuniovi/2023-simd/src/master/) (<https://bitbucket.org/2acuniovi/2023-simd/src/master/>) and [multi-threaded version](https://bitbucket.org/2acuniovi/2023-multi-thread/src/master/) (<https://bitbucket.org/2acuniovi/2023-multi-thread/src/master/>). The team must create two new private repositories from these public repositories following the same approach as when creating the repository of phase 1.

This phase is divided in two. A single-threaded program with SIMD extensions is written in the first part, while a multi-threaded program without these extensions is written in the second.

In this phase it is important to check that the resulting images obtained with the SIMD version and multi-threaded versions match the images obtained as a result in phase 1. You should use an application called diffImages (<https://www.atc.uniovi.es/grado/2ac/diffImages.tar.gz>) that calculates the difference between two images, displays them and their difference image. If both images are identical, a black image will be shown at the right. In addition, this difference is numerically expressed below as the mean and variance of pixels in the difference image.

2.2.1. SIMD program (5 points)

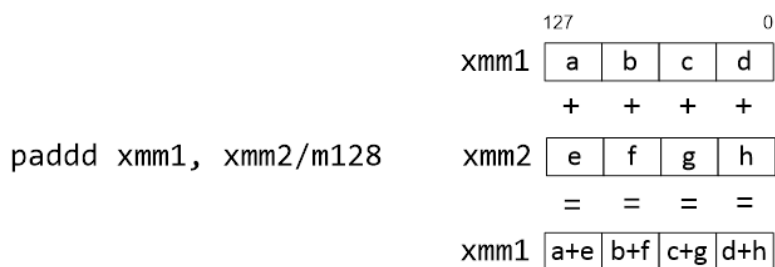
The program to write uses SIMD extensions of the architecture to improve performance. The project generating program `simd` must be used in this part. SIMD extensions will be used to improve the performance of the repetitive operations carried out to process images. These extensions are also called multimedia or vector extensions, since repetitive and simple operations on large vectors and matrices are commonly used in this field to encode images, audio or video.

Multimedia extensions are optional in the CPU architecture and they are grouped in instruction sets. Therefore, you will first check what SIMD extensions are supported in your CPU. As presented in the first lab session, you can use the `cpuinfo` file to check the complete list of instruction sets supported by your CPU.

You must check whether the CPU supports the following extensions, sorted from older to newer: MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AVX2 y AVX-512F. You do not need to check for other extensions. Notice that, support for newer extensions usually includes support for older ones.

Using SIMD extensions requires using intrinsics. These functions are similar to C functions, but each one is compiled to an assembly instruction.

For instance, intrinsic function `__m128i _mm_add_epi32 (__m128i a, __m128i b)` performs the addition of two vectors, `a` and `b`, each one containing 4 32-bit integers, and stores the result in another 4-integer vector. This instruction is translated into the `paddq xmm1, xmm2/m128` assembly instruction. Register `xmm1` is a 128-bit register used as source and destination. The second operand, `xmm2/m128`, is a 128-bit source register or a 128-bit operand stored in memory. The `xmm` registers were introduced with SSE2 extensions. This instruction performs the operation `xmm1+xmm2 → xmm1`, as shown below.



Special attention should be paid to memory alignment when dealing with intrinsics. For instance, accessing 128-bit operands must be done using memory addresses multiple of 16 (16 bytes are needed to store 128 bits, so 16 memory addresses are needed). The access to misaligned memory addresses may degrade performance and, in some microarchitectures, generate runtime errors.

To illustrate the usage of SIMD instructions, an example program is shown below. It adds 16 elements from a vector of 18 floats using packets of 256 bits. The code also shows the best approach to use multimedia extensions and avoid memory misalignments:

1. Declare constants to define the number of elements (*floats*) that can be stored in each packet of 256 bits.
2. Declare two variables of the same type as the packet to use: `__m256` in this case.
3. Initialize the source vectors for the operations, `a` and `b`.

4. Declare vector `c` aligned to the size of the packet to use. Its size must be enough to store all the data obtained by performing the operations. It is initialized to -1 values using intrinsics.
5. Since source vectors may be located in unaligned addresses, an intrinsic operation for loading unaligned data must be used. A temporal aligned variable is used for each vector.
6. The vector addition is performed on a data packet using the appropriate SIMD instruction.
7. Steps 5 and 6 are repeated for the next packet.


```

1  /*
2  * Main.cpp
3  *
4  * Created on: Fall 2022
5  */
6
7  #include <stdio.h>
8  #include <immintrin.h> // Required to use intrinsic functions
9
10
11 // TODO: Example of use of intrinsic functions
12 // This example doesn't include any code about image processing
13
14
15 #define VECTOR_SIZE      18 // Array size. Note: It is not a multiple of 8
16 #define ITEMS_PER_PACKET (sizeof(__m256)/sizeof(float))
17
18
19 int main() {
20
21     // Data arrays to sum. May be or not memory aligned to __m256 size (32 bytes)
22     float a[VECTOR_SIZE], b[VECTOR_SIZE];
23
24     // Calculation of the size of the resulting array
25     // How many 256 bit packets fit in the array?
26     int nPackets = (VECTOR_SIZE * sizeof(float)/sizeof(__m256));
27
28     // Create an array aligned to 32 bytes (256 bits) memory boundaries to store the sum.
29     // Aligned memory access improves performance
30     float *c = (float *)__mm_malloc(sizeof(float) * VECTOR_SIZE, sizeof(__m256));
31
32     // 32 bytes (256 bits) packets. Used to stored aligned memory data
33     __m256 va, vb;
34
35     // Initialize data arrays
36     for (int i = 0; i < VECTOR_SIZE; i++) {
37         *(a + i) = (float) i;          // a = 0, 1, 2, 3, â€¦
38         *(b + i) = (float) (2 * i);    // b = 0, 2, 4, 6, â€¦
39     }
40
41     // Set the initial c element's value to -1 using vector extensions
42     *(__m256 *) c = _mm256_set1_ps(-1);
43     *(__m256 *) (c + ITEMS_PER_PACKET) = _mm256_set1_ps(-1);
44     *(__m256 *) (c + ITEMS_PER_PACKET * 2) = _mm256_set1_ps(-1);
45
46     // Data arrays a and b must not be memory aligned to __m256 data (32 bytes)
47     // so we use intermediate variables to avoid execution errors.
48     // We make an unaligned load of va and vb
49     va = _mm256_loadu_ps(a);          // va = a[0][1]â€¦[7] = 0, 1, 2, 3, 4, 5, 6, 7
50     vb = _mm256_loadu_ps(b);          // vb = b[0][1]â€¦[7] = 0, 2, 4, 6, 8, 10, 12, 14
51
52     // Performs the addition of two aligned vectors, each vector containing 8 floats
53     *(__m256 *) c = _mm256_add_ps(va, vb); // c = c[0][1]â€¦[7] = 0, 3, 6, 9, 12, 15, 18, 21
54
55     // Next packet
56     // va = a[8][9]â€¦[15] = 8, 9, 10, 11, 12, 13, 14, 15
57     // vb = b[8][9]â€¦[15] = 16, 18, 20, 22, 24, 26, 28, 30
58     // c = c[8][9]â€¦[15] = 24, 27, 30, 33, 36, 39, 42, 45
59     va = _mm256_loadu_ps((a + ITEMS_PER_PACKET));
60     vb = _mm256_loadu_ps((b + ITEMS_PER_PACKET));
61     *(__m256 *) (c + ITEMS_PER_PACKET) = _mm256_add_ps(va, vb);
62
63     // If vectors va and vb have not a number of elements multiple of ITEMS_PER_PACKET
64     // it is necessary to differentiate the last iteration.
65
66     // Calculation of the elements in va and vb in excess
67     int dataInExcess = (VECTOR_SIZE)%(sizeof(__m256)/sizeof(float));

```

```

68
69 // Surplus data can be processed sequentially
70
71 for (int i =0; i< dataInExcess; i++){
72     *(c + 2 * ITEMS_PER_PACKET + i) = *(a + 2 * ITEMS_PER_PACKET + i) + *(b + 2 * ITEMS_PER_PACKET +
73 i);
74 }
75
76 // Print resulting data from array addition
77 for (int i = 0; i < VECTOR_SIZE; i++) {
78     printf("\nc[%d]: %f", i, *(c + i));
79 }
80
81 // Free memory allocated using _mm_malloc
82 // It has to be freed with _mm_free
83 _mm_free(c);
84
85 return 0;
}

```



You can get used to intrinsics by modifying the previous example increasing the size of the vector (VECTOR_SIZE) and processing it in a loop.

More information about intrinsics can be found in the following web page:

<https://software.intel.com/sites/landingpage/IntrinsicsGuide/>

In appendix SIMD references you can find several links that graphically explain how vector instructions work and some of the less common operations.

Intrinsics must be used in the cases where the same operation is executed on several items at the same time, as in the operations on images. For each of these operations, you must justify the intrinsic selected to reduce the response time as much as possible.

If the algorithm was repeated several times in the program of phase 1 to make the response time significant, **it must be repeated the same number of times** in this new version so the results can be comparable.

The **Release** version of the program must be run 10 times using the command-line interface, writing down the response times. Then, the average and the standard deviation of the response time of the program must be computed and written to the final document of the project. In addition, you must compute the speedup of this program compared to the original one. You must justify the achieved speedup.

2.2.2. Multi-threaded program (3 points)

The basic program (without SIMD instructions) should be converted into multi-threaded. To facilitate this task an appropriately configured project is provided to generate a program called `multi-thread`. This program is empty. The number of threads to use must match the concurrency level of the system. For example, if the computer has one processor with 4 cores with Hyperthreading, the maximum concurrency level is $1 \times 4 \times 2 = 8$, since the system may run 8 threads (without using time sharing). You have already worked with threads in Linux in lab session 1.2.

Notice that it is necessary to parallelize the algorithm, that is, you need to decompose the algorithm in smaller tasks that can be assigned to threads. For instance, the addition of the items of a $16 \times N$ vector can be divided in 16 additions of vectors of size N . Each of these additions can be carried out by a different thread. Finally, the main thread receives the partial additions of all the threads and combines them to compute the global addition.

As with the previous part, if the algorithm was repeated several times in the program of phase 1, it must be repeated the **same number of times** in this new version so the results can be comparable. Time measurements must include the time required to create and destroy the threads.

As with the previous programs, the **Release** version of the program must be run 10 times using the command-line interface, writing down the response times. Then, the average and the standard deviation of the response time of the program must be computed and written to the final document of the project. In addition, you must compute and justify the speedup of this program compared to the basic implementation.

3. Milestone

There is only one milestone during the project that will be assessed using this assessment matrix.

In this milestone you will have to deliver:

1. A short report containing:

- The information listed in the description of phase 2. The document must detail the work carried out by each team member and an estimation of the percentage it represents in the project.
- The detail of the algorithm used to filter the image.
- A screenshot of the original image before and after applying the filter for each of the three programs.
- A capture of using [diffImages](https://www.atc.uniovi.es/grado/2ac/diffImages.tar.gz) (<https://www.atc.uniovi.es/grado/2ac/diffImages.tar.gz>) to compare the image generated by the SIMD program with the image generated by the single-threaded program (reference).
- A capture of using [diffImages](https://www.atc.uniovi.es/grado/2ac/diffImages.tar.gz) (<https://www.atc.uniovi.es/grado/2ac/diffImages.tar.gz>) to compare the image generated by the multithreaded program with the image generated by the single-threaded program (reference).
- The response time and standard deviation measurements for each of the programs and the speedup of the SIMD and multithreaded programs against the reference program.
- Brief analysis of the results obtained.
- The work carried out by each team member and an estimation of the percentage it represents in the project.

2. The source code of the three programs.

The report and the source code will be compressed into a single file and posted in the group forum by one of its members.

Appendix A: Test images

A set of test images is provided to develop and test your programs. You can download all the images following these steps:



- Move to the **Pictures** directory in your home directory.

```
$> cd ~/Pictures
```

- Download the compressed file with all the images and uncompress them.


```
$> wget http://rigel.atc.uniovi.es/grado/2ac/2021-teamwork-images.tar.gz  
$> tar xvfz 2021-teamwork-images.tar.gz
```

Once you uncompress the file, several directories are created containing some images:

- `normal` . Contains images acquired with correct exposure time.
- `contrast` . Contains images acquired with low exposure times (-2) or high exposure times (+2). These images are suitable for algorithms that alter contrast or exposure time.
- `whitebalance` . Contains images acquired with white balance alterations. These are suitable for algorithms that correct white balance.
- `backgrounds` . Contains several backgrounds with the same size as the images to be used as the second image on fusion algorithms.

Appendix B: Visual Studio Code introduction

In order to develop the programs in the provided Linux virtual machine, Visual Studio Code will be used. This is a multiplatform IDE developed by Microsoft. This application is easy to use, but a brief guide is included here:

- Open the IDE. You must click in the  icon placed on the side bar of the desktop to open the IDE.
- Open a project. A project is located in a directory, so to open a project you must open a directory. Go to **File > Open Folder** and select the directory where your project is located.
- Edit. Select the `.cpp` file you want to edit. You can create new files or directories by pressing the corresponding button next to the project name.
- Debug. Go to **Run > Start Debugging** or press `F5` . The executable file will be created in the `debug` directory of the project.
- Add a breakpoint. Click next to the line of the program where you want to insert the breakpoint. A red point will appear.
- Variable inspection. By default, the debugger shows the value of all the variables of the program in the **Local Variables** side window.
- Add a variable to inspection. Right-click on the variable in the editor and select **Debug > Add to Watch**.
- Visual Studio Code supports version control using git. You may find help about how to use it in this [introduction to using git with Visual Studio Code](https://code.visualstudio.com/docs/editor/versioncontrol#_git-support) (https://code.visualstudio.com/docs/editor/versioncontrol#_git-support).

Appendix C: Assessment matrix

The table below will be used to assess the project. The quantitative weight of each item is not included, since some items are essential. For instance, a program cannot be assessed if it does not compile. This matrix should be used as a guide of the items to review before submitting the work, but not as a list of the tasks to complete.

Category	Description
Fundamental	It does not compile
	It crashes under normal running conditions

Category	Description
Algorithm	The filter formula is not implemented correctly
	It is not taken into account that the size of the images is not a multiple of the number of threads / packet size
	Out of range color components are accessed (be careful with threads and the last SIMD package)
	Some components are still to be processed (be careful with the threads and the last SIMD package)
	Color component saturation is not checked
	SIMD version does not correctly deals with aligned / unaligned memory
	SIMD version processes component by component instead of moving to the next packet (extra processing)
	In the SIMD version, a significant part of processing is carried out sequentially when it could be done using SIMD instructions

Category	Description
Logic	The number of repetitions must affect the entire algorithm, not just part of it (do not exclude thread / SIMD initializations)
	Response time incorrectly computed as the difference between the final and initial times
	Possible <code>clock_gettime</code> errors are not checked
	Magic numbers are used instead of constants (eg: number of repetitions)
	Errors are not checked when loading images (the program crashes if the image does not exist)
	The size of images is not verified to be the same
	Parameters are incorrect in thread creation functions (race conditions)
	Not all code to be measured (initializations) is included, or screen printing is included
	Common code in functions is not extracted (it is not parameterized)
	The image obtained in the SIMD and multi-threaded versions is different from that obtained in the initial version without SIMD extensions and implemented as a single-thread.
Code styling	Bad indentation
	Unsuitable identifier names
	Dirty or poorly commented code
Report	Some screenshots are missing
	Statistics are missing or miscalculated
	Acceleration is missing or miscalculated
	Other bugs in the report

Appendix D: SIMD references

- <https://software.intel.com/sites/landingpage/IntrinsicsGuide/> - Main reference for Intel Intrinsics. It shows the name of the intrinsic, its arguments and the returned result. It also shows the equivalent machine code and a description of the operation at bit level.
- <https://www.tommesani.com/MMXPrimer.html> - Graphical representations of MMX and SSE operations.

- <https://www.codeproject.com/Articles/874396/Crunching-Numbers-with-AVX-and-AVX> .- Using AVX for numerical computation. Sections 3.1 *Data Types* and 3.2 *Function Naming Conventions* explain data types and naming of intrinsics.
- <https://db.in.tum.de/~finis/x86-intrin-cheatsheet-v2.2.pdf> .- x86 intrinsics Cheat Sheet.