NYU School of Engineering
Convolution Unleashed

Iman Hosseini*

This report does include some code too, whenever it is part of the project to discuss them, but the full code will be attached in a zip file containing project files, on a windows machine it would be trivial to run them as each of them include project files for either Visual Studio 2019 or Android Studio or Vivado (so those IDEs know how to handle them).

1 Overview

1.1 Introduction

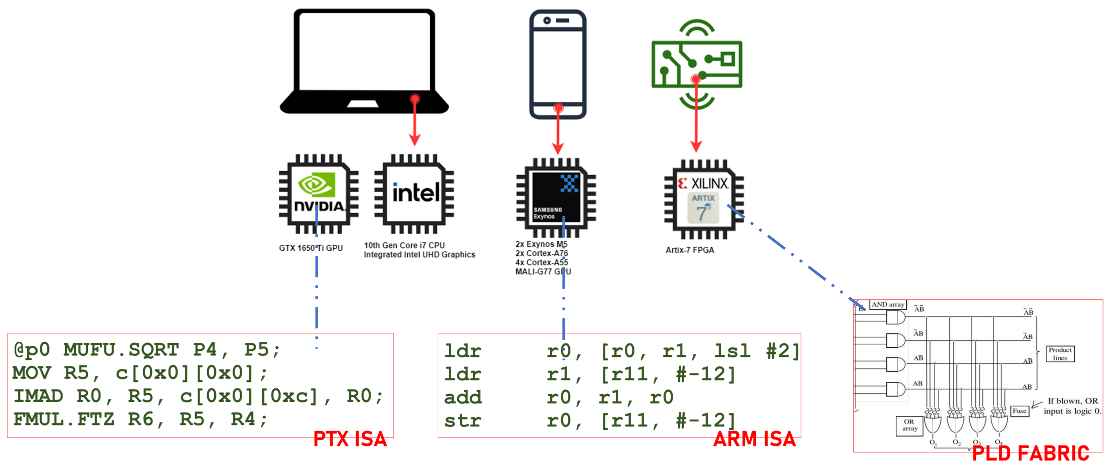
Computers have come a long way since the days of punch cards and gigantic mainframes, nowadays we are surrounded by microprocessors in various shapes and sizes. Looking at latest smartphone or laptop commercials it seems we should be in a golden age of superfast computers and the glitzy benchmarks speak of performance in the range of GFLOPs[1] even for midrange pcs, and yet when I was waiting for my python code for a convolution impatiently tapping my fingers, I had to ask myself this cannot be the golden age of computers, this shouldn't take this much!

The thing with benchmarks and commercials, is that they are about *peak performance*, about what is theoretically *possible* and more often than not, that is far from what actually happens. Typically we pick an algorithm, pick a platform and focus on the high level issues of an algorithm, parameter tuning etc., but in this project I aim to do something very different, I take the most simple task (2d convolution) and instead of high level explorations, I explore the various ways to implement that across a range of hardware and software platforms, and focus on how easy it is to approach the promised *peak performance*, and the comprises in going for more performance. One of the complicating issues of the modern landscape is that there are many different devices with inherently different architectures, and the field is crowded in software platforms as well, with different companies rolling out their SDKs, some are for specific devices (like CUDA for NVIDIA devices), and some have been attempting to unify the platforms for different devices, ultimately aiming for a *write-one use anywhere* approach. (like OneAPI advocated by Intel)

*shz230@nyu.edu

[1]Giga Floating Point Operation per second

Despite the simplicity of convolution, it is a very prominent algorithms which many other vision algorithms build on. It is also an example which has the features of many vision algorithms, and so convolution would be a window into judging how vision algorithms in general can be implemented with performance in mind, over these various architectures. For this work I asked myself with the usual modern household devices (and an FPGA!), how fast can I get a 128x128 image convolved with a 3x3 filter? So my devices where an Arty-7 FPGA board, a Dell XPS laptop and a Samsung galaxy note, where the smartphone and the laptop in fact have 2 possible chips for the task, the CPU and the GPU and I evaluated both. Also I implemented our test algorithm using ARM Neon Extensions and x86 AVX Extensions, which are architecture specific vector extensions promising higher performance for parallel tasks such as convolution.



```
@p0 MUFU.SQRT P4, P5;
MOV R5, c[0x0][0x0];
IMAD R0, R5, c[0x0][0xc], R0;
FMUL.FTZ R6, R5, R4;
```

```
ldr     r0, [r0, r1, lsl #2]
ldr     r1, [r11, #-12]
add     r0, r1, r0
str     r0, [r11, #-12]
```

Figure 1: FPGA is fundamentally different

1.2 2D-Convolution

In this work, for all intents and purposes, convolution *is equivalent* for us to correlation: in terms of calculations they only differ in reversion (flipping the axes) of the filter/kernel. Also optimizations for separable kernels (as it is possible for gaussian blur for example) or FFT optimizations is not our concern here and the calculations are done as-is. The calculation is to slide the kernel over the image (possibly with zero-padding to ensure same dimensions for output and input) and for each pixel in the output g given a filter k and image f we have:

$$g(x,y) = k * f(x,y) = \sum_{dx=-a}^{dx=a} \sum_{dy=-b}^{dy=b} k(dx,dy) f(x+dx, y+dy)$$

where a, b are dimensions of the kernel, and you can think of f as the padded image. The variations to padding can be to use the edge values for the padding on every side, or even based on situation toroidal-padding, where we identify the edges on the opposite side such that for example $f(-x, y) = f(h - 1 - x, y)$, (where h is image height) with the namesake being that this effectively implies that the toplogy of the image is a torus (think old-school arcade games, where you would reappear on the left when you crossed the right side of the screen).

1.3 CPU vs GPU vs FPGA

Intel unveiled Pentium in 1993, and the x86 architecture reigned supreme in the world of CPUs, and at the time GPUs did not exist. With more demand for high fidelity CGI, and higher throughput for numerical calculations, the idea of accelerator chips was touted and in 1999 the first ever GPU[2], Nvidia GeForce 256 was unveiled. The idea was that x86 architecture did not really allow for exploiting parallel nature of many numerical calculations (prominent in CGI), it works it way over a series of instructions sequentially, and even though it was running at high frequencies and was really good at sequential computation, many problems are parallel in nature and there was room for much higher performance. The x86 ISA (Instruction Set Architecture) supported a variety of instructions, including complex ones, but for a typical rendering algorithm you really only need a few basic instructions which are not sequentially related and can be run simultaneously and this is what GPUs were made for.

Another development that we will come back to is vector extensions. GPUs were not the only answer to more parallel workloads and the quest for higher throughputs, in 1997 intel came up with the MMX extensions to the original x86 ISA, which added support for a new set of instructions which could do multiple operations at once. Whereas the GPUs solution was to have multiple simple cores which could process a stream of instructions at once, MMX had another approach: what if we still had our one (or a small number of cores) core but it was now capable of executing a **ADD** instruction which adds multiple numbers at once. To achieve this task it added 8 64bit registers named **MMX0** to **MMX7** which could hold 2 32bit number or 4 16bit number or 8 8bit number.

At the same time, research was underway to make logic devices which were programmable. Xilinx had made the first FPGA in 1985 and the Navy was working on making a device with 600,000 reprogrammable gates, which succeeded in 1992. The idea of FPGAs is that, unlike the CPUs and GPUs which are ASIC (Application Specific Integrated Circuit) and thus are static in terms of the logic circtcuitry on the chip, FPGAs are what you want them to be: you can program your logic circuits on them. FPGAs allow for high energy efficiency and low latency, and another promising feature is that manufacturing ASICs is only viable

[2]pedantically there did exist similar chipsets before that but this was really the point where GPUs as we know it were born

in large numbers and it is a very involved, time consuming task, and thus the smallest of errors is very costly. Imagine finding a bug in your chip after multiple months that it took to manufacture, and now you are left with thousands of useless devices that you cannot ship to customers. FPGAs allow for an easier and faster iterations to try out various designs (CPU designs are tested on FPGA for example). Regarding their low latency, a CPU has a rather complex ISA, it should be able to handle many things as it is a general purpose device, and as everything is burned on the chip permanently you'd have some extra instructions in there even if those instructions are only used by a small number of applications/customers, but imagine the case that you have a fighter jet and you only want to run an edge detection algorithm to find targets and that is all, an FPGA allows you to implement just enough logic circuitry for that case, and so you are free of all the overheads: when a CPU is working, it has to fetch instructions, decode them, and execute them (sequentially), each step taking time as for example x86 instructions can be anywhere between 1 to 15 bytes and so it is not trivial to even decode the instructions, but the FPGA is just the circuitry for that task (i.e. edge detection) and it does not need to do the additions/multiplications sequentially, it can go over many of them at the same time, without any overhead.

To recap, GPUs are capable to excel at highly parallel tasks, CPUs are made for general purpose cases and *try* to also do better at parallel tasks using ISA extensions, and FPGAs promise reconfigurability, low latency and energy efficiency.

1.4 RISC vs CISC

There used to be 2 camps when it comes to CPU development: Reduced Instruction Set Computing and Complex Instruction Set Computing. A CPU is a bunch of logic gates (think ANDs and ORs) burnt onto a piece of silicon which implements an ISA, this ISA is the interface with which the CPUs can be ordered to solve our problems. A CPU has access to RAM as the main memory and other (slower) storage devices, when we run a program, it gets loaded from our storage device (SSD or HDD) into the RAM and then the CPU starts executing the instructions, the ISA is thus what we have to express our algorithms in, actually the days of low level development like that are long gone, so actually it is the compiler that generates the instructions for our program from a higher level, probably architecture agnostic, form (like C). The question then is should we design the ISA to consist of complex instructions (which promises more expressibility) or should we design it with the smallest simplest of instructions. The argument for CISC, is that we can have 1 complex instruction for a task that would take 3 RISC instructions, while RISC people point out the fact that sure we can have complex instructions, but it is not free, more complex instructions means that decoding them would take more time, implementations would get slower whereas with RISC those simpler instructions can be made to run fast.

The discussion in above paragraph is a simplified one though, and the exact definition of these terms is not really clear-cut as that, for example there are RISC ISAs which have as much instructions as CISC designs, so another way to characterize for RISC is to say that in RISC each instruction *does less* rather than just having a smaller number of instructions. Modern example of RISC is the ARM architecture used in (mostly) mobile devices, due to the importance of battery life in mobile devices (tablets/smartphones) and ARM excels in this, but it is a very interesting time for ARM as the use of M1 chip in new MacBooks shows that the perceived lack in performance may not be the case anymore. One advantage of ARM is the licensing and how other vendors can get the license and customize ARM cores to their liking, and another RISC player is RISC-V which takes an step even further, being open and free to use. On the other side, x86 architecture is the CISC player which reigns supreme in server and pc / high performance laptop sectors, though this advantage is dwindling. Also another thing to note is that even though x86 is CISC, throughout it's lifetime it has adopted ideas from RISC, and under the hood it is really RISC: the ISA that we are exposed to is still CISC but what happens inside is that those instructions get translated into micro operations (μOps) so 1 instruction might get translated into multiple micro operations and this is in fact the main idea of RISC, the design of that simpler ISA-within-an-ISA (which by the way, is not transparent to the user) is RISC.

While trying out assembly programming one of the first changes you notice between ARM and x86, is that in ARM there are no instructions for memory-to-memory move, while in x86 you can do that. Another issue with RISC is that due to the lack in expressiveness compared to x86, programs would take more instructions and thus more memory ARM added a 16bit set of instructions called Thumb to make up for it. The way ISA extensions in ARM work is more extensible than x86, and they act via *mode switches* so that there are instructions which switch between modes (for example from AArch32 to Thumb), for each mode (which you can think of as an ISA on itself) the instructions are fixed-length, this means decoding the instructions is faster.

1.5 Registers On ARM & x86

First there was 8bit 8080 then 8086 came being a 16bit extension of that, then extended to 32 and later x86_64, and so on both modern x86 and ARM cores there are 64bit registers. In ARM the registers are named $R0$ to $R15$ with $CPSR$ being the extra "Current Program Status Register", in x86 there is AX, BX, CX, DX where each of them is the 16bit names and to access the lower and higher 8bits respectively they are named AL, AH for AX (same with the other three) and they become EAX if we are to use them with 32bits, and RAX to use them as 64bits, in addition to them there is $R8$ to $R15$ which are 64 bit, and they become suffixed with D or W or B to address 32bit, 16bit or 8bits of them, and we have RBP , RSP, RDI, RSI (addressed same as the ABCD registers but they do not have equivalent for AH to get the higher 8bits), for ARM it is easier, wherever we are in 32bit mode Ri would be 32bit as it should. With vector extensions, x86 also has $XMM0$ to $XMM15$ which are 128bits each, later $YMM0$ to $YMM15$ was added which are 256bits and AVX512 adds $ZMM0$ to $ZMM31$ which are 512bits (but my CPU and many others don's support AVX512). ARM neon extensions add 16 128bit registers $Q0$ to $Q15$ and 32

64bit ones $D0$ to $D31$.

2 Dell XPS

2.1 the setup

The machine comes with an 8 core Intel Core i7 10875H Comet Lake CPU which supports all the extended x86 instructions (MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, AES, AVX, AVX2, FMA3) which we will later see more of, and has 32KB of L1 cache, and 256 KB of L2 cache, with 16MB of L3, which we will also discuss more later. It also has integrated graphics (Intel UHD) and Nvidia discrete GPU GTX 1650 Ti with 4 GB of memory. So we to run a convolution calculation all these 3 devices are capable for crunching numbers for us.

2.2 Caches, Branches & Port Contention

The general idea of caching is to keep most used data in faster memory, and so starting from the slowest memory, and moving towards faster memory we have a pyramid like shown in figure 2. L4 Cache is not available on all systems (like our system), and L3 is shared across cores where L1,L2 are not, also L1 caches are actually 2 cache one for data and one for instruction.

To have an accurate realistic measurement of the timings here, the Intel MLC tool[3] can be used and so for our machine we have the numbers shown in 3. One of the first issues in performance can now be seen here: caches are not byte-by-byte, when we access a new memory location a whole cache line is brought into the cache and when there is cache eviction a whole cache line is evicted, if we keep bringing in data from far-off places we pollute the cache and we end up moving around data across cache levels every time. A rough calculation shows for a very-small image which fits in the L1 cache we could have all memory accesses from L1, but that is really rare but even for a large image, due to the nature of how convolution is calculated, with the kernel slid over the image, the locality of the data means that we will still have high cache hits, if we have a decent implementation which does not pollute the cache.

Another feature of modern processors is branch prediction. As early as 1995 x86 cores where using a trick called out-of-order execution to achieve higher performance, the idea is closely related to pipelining: the complexities involved with instructions meant that it could take as much as 15 clock cycles for execution of an instruction, with multiple stages, the pentium pro pipeline can be seen in figure 6 (from Agner Fog's microarchitecture docs available freely[4]). What the pipeline does is that it makes it possible to do multiple things

[3]https://software.intel.com/content/www/us/en/develop/articles/intelr-memory-latency-checker.html

[4]https://www.agner.org/optimize/microarchitecture.pdf

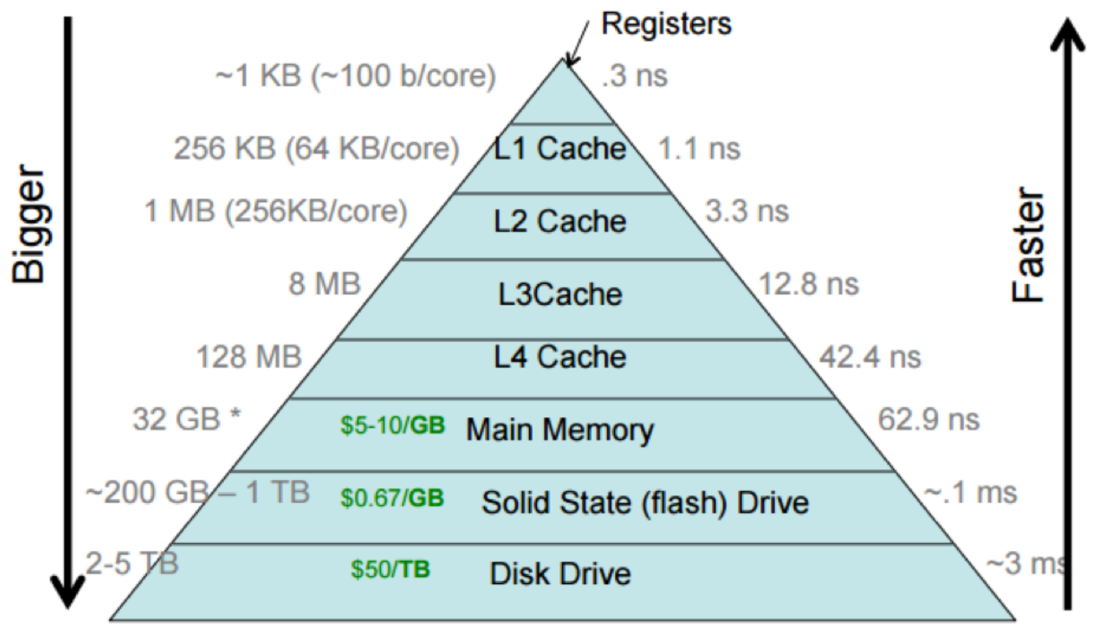
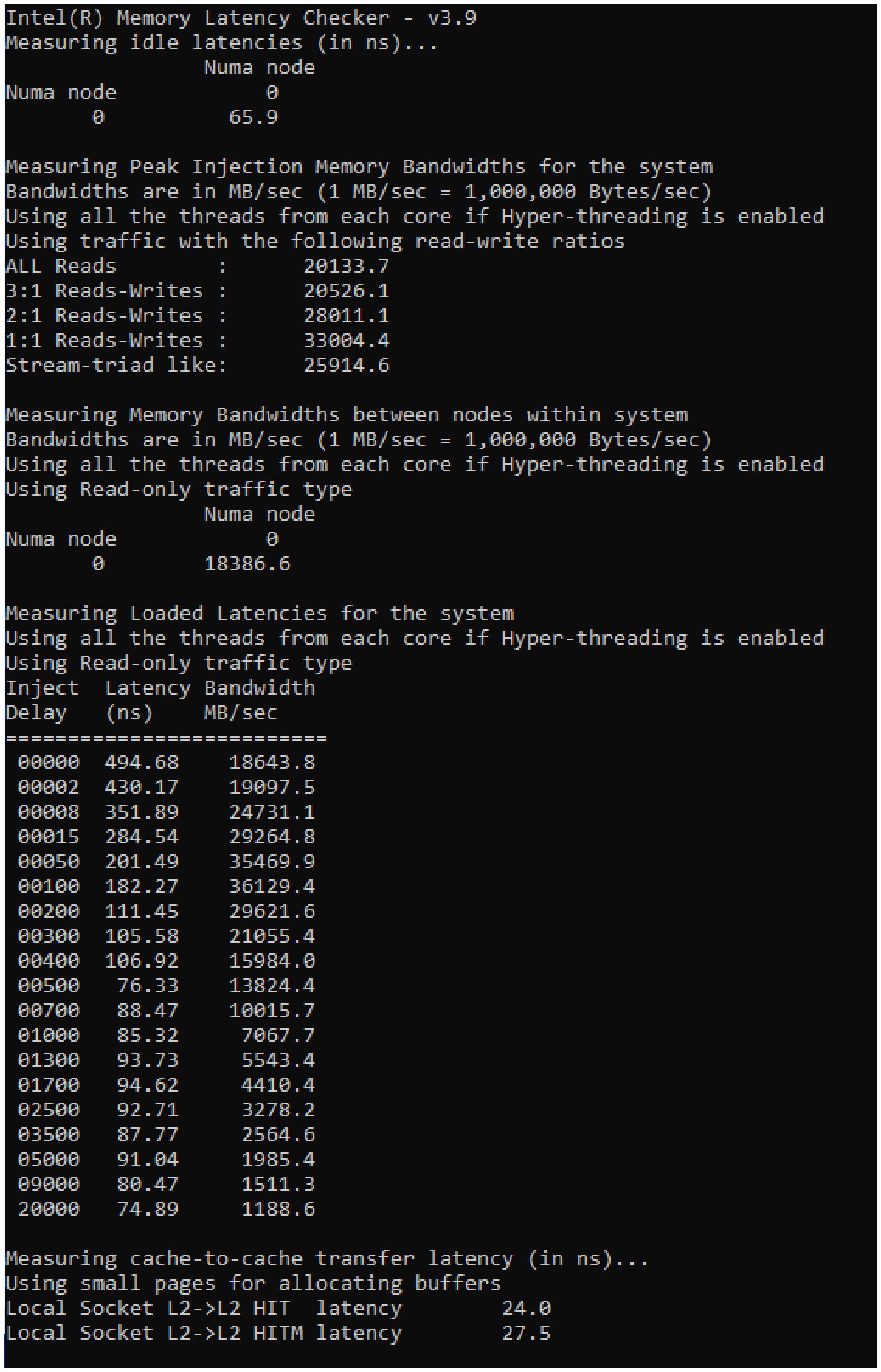


Figure 2: the memory pyramid

at once and keep different parts of the processor busy: while executing some arithmetic, why not start fetching the next instruction? And then out-of-order allows doing things even in another order than the instructions are issued: if instruction i is stalled but $i + 1$ can be executed why not execute it? These tricks work as long as the later instructions have no dependency on stalled instructions, so a long chain of dependent instructions would mean that effectively we have to still go over them sequentially, and also another issue is that the branching instructions mean that we would not be even sure what the next instructions are, before executing the jump, and to mitigate this issue a branch predictor unit tries to predict whether a jump is taken or not, and based on the prediction starts executing the instructions, but if the prediction was wrong, it has to dial back the partially executed instructions. The last stage in the pipeline is called retirement, the instructions dialled back due to branch misprediction are not retired, so ideally we would have all (or most) of the instructions which go into the pipeline end up retired.

Finally the last performance issue to mention regarding processors (mostly x86 in mind) is port contention and register spills. A processor can be divided into the front-end: in charge of fetching and decoding instructions, and a back-end, where the execution happens, the back-end has a number of execution units (also called ports): the instructions as mentioned earlier actually become chopped into microOps (micro-instructions) and those instructions get into a queue and are then sent to scheduler ports, but not all ports are capable of scheduling all instructions. (and these are *per-each-core*) On our core, there are

7 ports which can do tasks as shown in figure **??**, if there are more microOps of a certain type than there are ports, some of them have to wait, and so port-contention can also be a cause for slowdown. Also the backend does register renaming, which abstracts logical registers from physical registers, this can help reduce some data-dependencies: imagine having $R1 \leftarrow M[x]$; $R1 \leftarrow R1 + 1$; $M[x] \leftarrow R1$; $R1 \leftarrow M[y]$; $R1 \leftarrow R1 + 1$; $M[y] \leftarrow R1$; this chain of instructions are all dependent, but they don't have to, you can use $R2$ in the last 3 instructions and then they do not have to wait up on $R1$, the register renaming allows this to happen in hardware level, so the registers we see are not necessarily the same ones used eventually in hardware.



```
Intel(R) Memory Latency Checker - v3.9
Measuring idle latencies (in ns)...
                Numa node
Numa node            0
       0          65.9

Measuring Peak Injection Memory Bandwidths for the system
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)
Using all the threads from each core if Hyper-threading is enabled
Using traffic with the following read-write ratios
ALL Reads        :      20133.7
3:1 Reads-Writes :      20526.1
2:1 Reads-Writes :      28011.1
1:1 Reads-Writes :      33004.4
Stream-triad like:      25914.6

Measuring Memory Bandwidths between nodes within system
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)
Using all the threads from each core if Hyper-threading is enabled
Using Read-only traffic type
                Numa node
Numa node            0
       0        18386.6

Measuring Loaded Latencies for the system
Using all the threads from each core if Hyper-threading is enabled
Using Read-only traffic type
Inject  Latency Bandwidth
Delay   (ns)    MB/sec
==========================
 00000  494.68    18643.8
 00002  430.17    19097.5
 00008  351.89    24731.1
 00015  284.54    29264.8
 00050  201.49    35469.9
 00100  182.27    36129.4
 00200  111.45    29621.6
 00300  105.58    21055.4
 00400  106.92    15984.0
 00500   76.33    13824.4
 00700   88.47    10015.7
 01000   85.32     7067.7
 01300   93.73     5543.4
 01700   94.62     4410.4
 02500   92.71     3278.2
 03500   87.77     2564.6
 05000   91.04     1985.4
 09000   80.47     1511.3
 20000   74.89     1188.6

Measuring cache-to-cache transfer latency (in ns)...
Using small pages for allocating buffers
Local Socket L2->L2 HIT  latency        24.0
Local Socket L2->L2 HITM latency        27.5
```

Figure 3: memory measurement

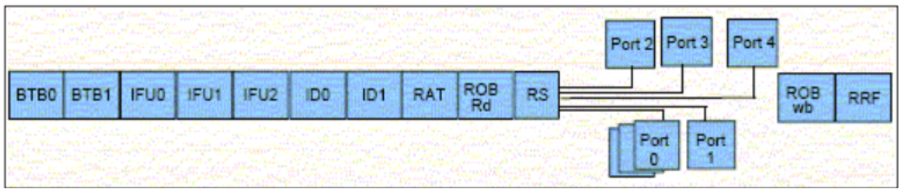


Figure 6.1. Pentium Pro pipeline.

The pipeline is illustrated in fig. 6.1. The pipeline stages are as follows:

BTB0,1:	Branch prediction. Tells where to fetch the next instructions from.
IFU0,1,2:	Instruction fetch unit.
ID0,1:	Instruction decoder.
RAT:	Register alias table. Register renaming.
ROB Rd:	μop re-ordering buffer read.
RS:	Reservation station.
Port0,1,2,3,4:	Ports connecting to execution units.
ROB wb:	Write-back of results to re-order buffer.
RRF:	Register retirement file.

Figure 4: Pentium Pro Pipeline

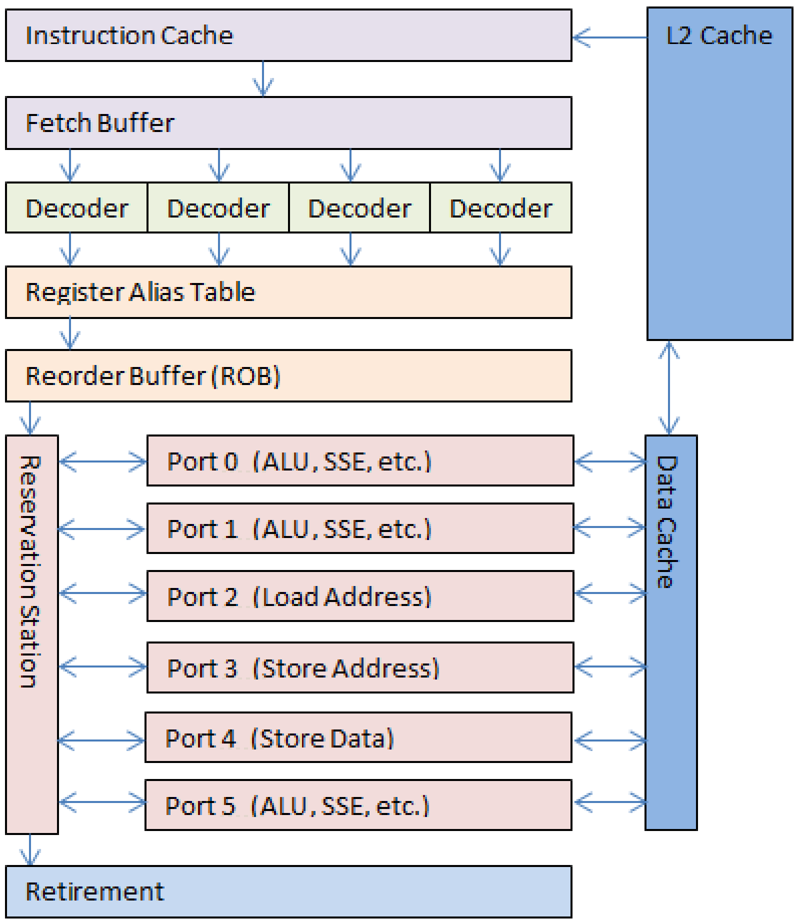
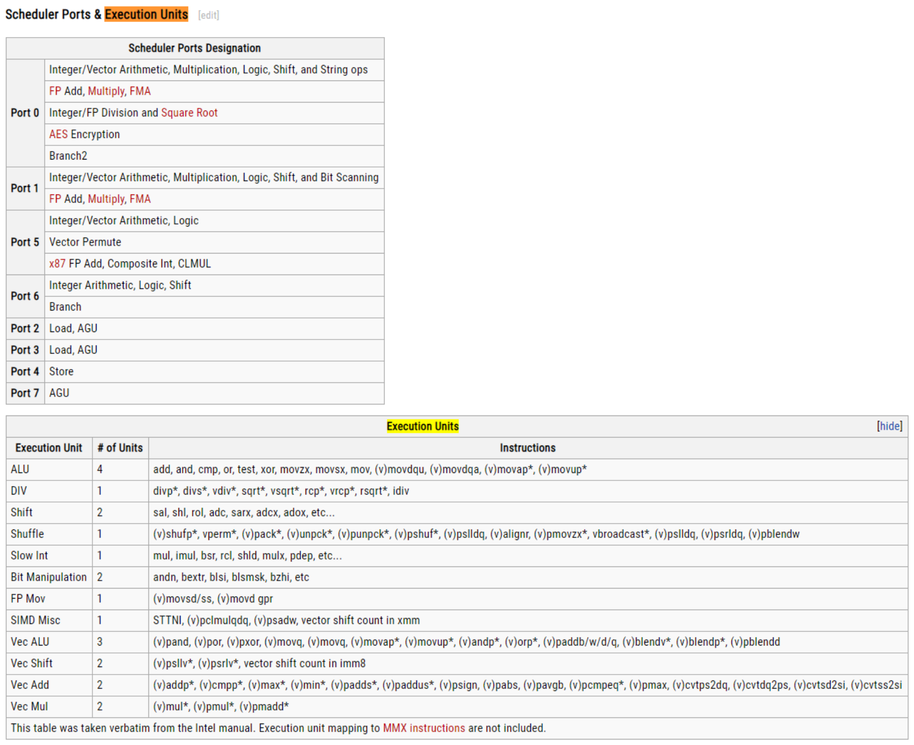


Figure 5: cpu does a lot!



Scheduler Ports & Execution Units [edit]

Scheduler Ports Designation	
Port 0	Integer/Vector Arithmetic, Multiplication, Logic, Shift, and String ops
	FP Add, Multiply, FMA
	Integer/FP Division and Square Root
	AES Encryption
	Branch2
Port 1	Integer/Vector Arithmetic, Multiplication, Logic, Shift, and Bit Scanning
	FP Add, Multiply, FMA
Port 5	Integer/Vector Arithmetic, Logic
	Vector Permute
	x87 FP Add, Composite Int, CLMUL
Port 6	Integer Arithmetic, Logic, Shift
	Branch
Port 2	Load, AGU
Port 3	Load, AGU
Port 4	Store
Port 7	AGU

Execution Units [hide]

Execution Unit	# of Units	Instructions
ALU	4	add, and, cmp, or, test, xor, movzx, movsx, mov, (v)movdqu, (v)movdqa, (v)movap*, (v)movup*
DIV	1	divp*, divs*, vdiv*, sqrt*, vsqrt*, rcp*, vrcp*, rsqrt*, idiv
Shift	2	sal, shl, rol, adc, sarx, adcx, adox, etc...
Shuffle	1	(v)shufp*, vperm*, (v)pack*, (v)unpck*, (v)punpck*, (v)pshuf*, (v)pslldq, (v)alignr, (v)pmovzx*, vbroadcast*, (v)pslldq, (v)psrldq, (v)pblendw
Slow Int	1	mul, imul, bsr, rcl, shld, mulx, pdep, etc...
Bit Manipulation	2	andn, bextr, blsi, blsmsk, bzhi, etc
FP Mov	1	(v)movsd/ss, (v)movd gpr
SIMD Misc	1	STTNI, (v)pclmulqdq, (v)psadw, vector shift count in xmm
Vec ALU	3	(v)pand, (v)por, (v)pxor, (v)movq, (v)movq, (v)movap*, (v)movup*, (v)andp*, (v)orp*, (v)paddb/w/d/q, (v)blendv*, (v)blendp*, (v)pblendd
Vec Shift	2	(v)psllv*, (v)psrlv*, vector shift count in imm8
Vec Add	2	(v)addp*, (v)cmpp*, (v)max*, (v)min*, (v)padds*, (v)paddus*, (v)psign, (v)pabs, (v)pavgb, (v)pcmpeq*, (v)pmax, (v)cvtps2dq, (v)cvtdq2ps, (v)cvtsd2si, (v)cvtss2si
Vec Mul	2	(v)mul*, (v)pmul*, (v)pmadd*

This table was taken verbatim from the Intel manual. Execution unit mapping to MMX instructions are not included.

Figure 6: scheduling ports from wikichip

What we discussed shows that modern CPUs are very complicated, and reasoning about performance is a multi-faceted complex issue and more often than not, even knowing what actually is happening is not even possible unless we actually *measure* as it is hard to speculate on what the CPU is doing (and this issue is *after* we have reasoned about what the compiler has done to our code).

2.3 Python

The easiest implementation possible: using numpy to make an array the size of the input, and then filling it pixel by pixel, based on the 9 pixels in the sliding window (remember the kernel is 3x3):

```
def conv(img):
    res = np.zeros(img.shape,dtype=np.float)
    for (i,j),_ in np.ndenumerate(res):
        aux = 0.
        choices = [(i-1,j-1), (i,j-1), (i+1,j-1), (i-1,j), (i,j), (i+1,j),
        ↪  (i-1,j+1), (i,j+1), (i+1,j+1)]
        for i,ch in enumerate(choices):
            x,y = ch
            if y>=img.shape[1] or y<0 or x>=img.shape[0] or x<0:
                continue
            aux += krnl[i]*img[x,y]
        res[i,j] = aux
```

The code is very simple, we go over the pixels, and skip over those pixels that are out of bounds (effectively means we are padding with 0), for our tests that would be 128x128 images and 3x3 kernel (gaussian filter), this takes 302359 (us) (or roughly 302 (ms)) which as we will soon see, is VERY bad. If we use the library code available (now the code becomes a one-liner, just calling that function) in *scipy.ndimage.filters* called *gaussian_filter* this time goes down by 2 orders of magnitude to roughly 1000 (us). This is very surprising, why such a great chasm? To see what is happening lets breakdown what is happening: Python is an interpreted language which means that the Python process running on our system is a program which goes through our program and *interprets* it and executes the relevant python bytecode (like Java which has JVM python also has a stack-based VM) this is a big overhead, a ”+” in python does not become an x86 ”ADD” instruction, it becomes an add in python VM, and then there is some x86 loop (inside the python.exe process) which goes over the python bytecode, reaches that ”add” python-vm instruction and then exectues the subroutine for the python-vm ”add” instruction (which does have an x86 ADD somewhere + other stuff), also you should avoid *np.ndenumerate* as the plague: it is very slow. While the scipy function uses actual fast kernels in native code, and so when you call *gaussian_filter* it ends up calling up to some fast native code which is much faster than the whole python-vm way, you can see those kernels (which are coded in C) in the scipy github repository here [5].

To see the python bytecode instructions that a function translates to we can use dis, the output also includes line numbers so that we can check what each line becomes.

```
import dis
print(dis.dis(conv))
--------------------------- aux += krnl[i]*img[x,y]
 19      >>  224 LOAD_FAST                  5 (aux)
             226 LOAD_GLOBAL                6 (krnl)
             228 LOAD_FAST                  2 (i)
             230 BINARY_SUBSCR
             232 LOAD_FAST                  0 (img)
             234 LOAD_FAST                  8 (x)
             236 LOAD_FAST                  9 (y)
             238 BUILD_TUPLE                2
             240 BINARY_SUBSCR
             242 BINARY_MULTIPLY
             244 INPLACE_ADD
             246 STORE_FAST                 5 (aux)
             248 JUMP_ABSOLUTE            162
         >>  250 POP_BLOCK
```

A key point we learn here is that Python is very slow, yet it shouldn't be ruled out because if we use libraries, the computationally intensive parts hold up as those parts use fast kernels (here by kernel, we mean *computationally intensive functions*) implemented in C and compiled to fast native code. And this can also be done (albeit it is hard) by us too through FFI (Foreign Function Interface) which let's us call to outside code from python or using python builtins.

2.3.1 C

Our next attempt is to use C:

```
void conv2d_d(const float* img, const float* kernel, float* result) {
    for (int i = 0; i < H; i++) {
        for (int j = 0; j < W ; j++) {
            int itop = i*S+j;
            int i_img[9]{itop, itop+1, itop+2, itop+S, itop+S+1, itop+S+2,
            ↪  itop+2*S, itop+2*S+1, itop+2*S+2};
            float v = 0.0f;
            for (int k = 0; k < 9; k++) {
```

[5]https://github.com/scipy/scipy/tree/701ffcc8a6f04509d115aac5e5681c538b5265a2/scipy/ndimage/src

```
                v += img[i_img[k]] * kernel[k];
            }
            result[i * W + j] = v;
        }
    }
}
```

This is almost a line-by-line translation from our python code. And it achieves a remarkable speed-up to roughly 350 (us). This speedup is mostly due to the shedding of all python abstractions (even calling to native code from python is not free and has an overhead).

2.4 Intel MKL

Intel oneAPI is the latest attempt by Intel to allow access to fast kernels and unify various legacy APIs doing the same in one library. Intel launched MKL (Math Kernel Library) in 2003, with the idea being to allow access to fast functions in statistics, vector math, differential equations and deep neural networks for Intel users, these kernels would be optimized by Intel which probably would know their CPUs better than anyone and also be give the customers an edge. Other efforts were also implemented like the IPP (Integrated Performance Primitives), Intel's own compilers, TBB (Thread Building Blocks) as a faster better thread library and so on. With oneAPI they brought all those disjoint libraries together, opensourced the code, and also try to allow for inter-operability with intel FPGA and even CUDA (so that you can code in DPC++ API and then translate that to CUDA code), even numpy does use MKL under the hood (albeit previous versions of MKL it seems) you can see the details by looking at *np.__config__*.

The development experience was not so smooth, the documentation was not great and I had to ask for clarifications on the Intel developer forum but they were quick to lend a hand and I managed to make this work. MKL_INT is a type to hold indices and sizes to pass to MKL functions, and a VSLConvTask is made by specifying the sizes of the kernel, image and strides, and passing pointers to each and executing it. This is in fact equivalent to using a prepared function like we did with SciPy, in that we do not do the actual convolution calculation, yet it is very verbose, but all that verbosity is just to specify the task (which is based on the MKL documentation). The result ended up being less than promising, 18920 (us) if we compiled with parallel flag and 4479 (us) for Sequential. The Parallel section is of course not suited for us, as it is thread safe, but we are only running on a single thread here.

```
int main(){
    VSLConvTaskPtr task;
    MKL_INT f_shape[] = { 128, 128 };
    MKL_INT g_shape[] = { 3, 3 };
    MKL_INT Rmin[] = { 0, 0 };
    MKL_INT Rmax[] = { f_shape[0] + g_shape[0] - 1, f_shape[1] + g_shape[1]
     ↪  - 1 };
    MKL_INT h_shape[] = { Rmax[0], Rmax[1] };
    MKL_INT h_start[] = { 0, 0 };
    MKL_INT f_stride[] = { f_shape[1], 1 };
    MKL_INT g_stride[] = { g_shape[1], 1 };
    MKL_INT h_stride[] = { h_shape[1], 1 };
    float* f = new float[f_stride[0] * f_shape[0]];
    float* g = new float[g_stride[0] * g_shape[0]];
    float* h = new float[h_stride[0] * h_shape[0]];
    for (int i = 0; i < f_shape[0]; ++i)
        for (int j = 0; j < f_shape[1]; ++j)
            f[i * f_stride[0] + j] = 10;
    for (int i = 0; i < g_shape[0]; ++i)
        for (int j = 0; j < g_shape[1]; ++j)
            g[i * g_stride[0] + j] = 10;
    memset(h, 0, sizeof(h[0]) * h_stride[0] * h_shape[0]);
    int status;
    status = vslsConvNewTask(&task, VSL_CONV_MODE_AUTO, 2, f_shape,
     ↪  g_shape, h_shape);
    assert(status == VSL_STATUS_OK);
    status = vslConvSetStart(task, h_start);
    assert(status == VSL_STATUS_OK);
    auto t1 = std::chrono::high_resolution_clock::now();
    status = vslsConvExec(task, f, f_stride, g, g_stride, h, h_stride);
    assert(status == VSL_STATUS_OK);
    auto t2 = std::chrono::high_resolution_clock::now();
    auto ms_int = std::chrono::duration_cast<std::chrono::microseconds>(t2
     ↪  - t1);
    std::cout << "exec time: " << ms_int.count() << "us\n";
    status = vslConvDeleteTask(&task);
    assert(status == VSL_STATUS_OK);
    for (int i = 0; i < h_shape[0]; ++i)
    {
        printf("%3i: ", i);
        for (int j = 0; j < h_shape[1]; ++j)
        {
            printf("%4.0f ", h[i * h_stride[0] + j]);
        }
        printf("\n");
    }
    return 0;
```

```
}
```

2.5 AVX

We can use AVX instructions ourselves, now actually writing out a ".s" file and assembling and linking it by hand is not necessary (and not wise) because compilers have simple wrappers around those instructions called *Compiler Intrinsics* and to see the details of instructions there is an online guide website[6]. We have to include "immintrin.h" to use them, and then based on what we want to do, we call the functions. Recall that we have access to 256bit registers, and these can hold 8 32bit integers, or 8 floats, to look into these registers based on how we are using them, there are union definitions inside the header like this:

```
typedef union __declspec(intrin_type) __declspec(align(32)) __m256 {
    float m256_f32[8];
} __m256;

typedef struct __declspec(intrin_type) __declspec(align(32)) __m256d {
    double m256d_f64[4];
} __m256d;

typedef union  __declspec(intrin_type) __declspec(align(32)) __m256i {
    __int8              m256i_i8[32];
    __int16             m256i_i16[16];
    __int32             m256i_i32[8];
    __int64             m256i_i64[4];
    unsigned __int8     m256i_u8[32];
    unsigned __int16    m256i_u16[16];
    unsigned __int32    m256i_u32[8];
    unsigned __int64    m256i_u64[4];
} __m256i;
```

The main part of our code would be to multiply the kernel values with the pixel values, with these extensions we can put 8 of the pixels in one 256bit register, and the relevant kernel values in another, and then multiply them in a single instruction and then the result would be 8 values tucked into a single register, then we need a way to add those 8 values together, add the extra pixel*kernel (256bit registers support 8-wide calculations but our kernel is 3x3 and needs 9) to it and we would have 1 output pixel. Another way, which is faster, is that instead of going through the output pixels, calculating them 1 by 1, we go through them 8 by 8, i.e. to track them we hold $p1 * k1$ in 1 register, for 8 of the output pixels, so it will hold the upper left part of the 3x3 kernel times the upper-left pixel value

[6]https://software.intel.com/sites/landingpage/IntrinsicsGuide/

for pixels i thorugh $i+7$ in register $v1$ (we call it $v1$ in our code, and leave the actual register used from the YMM registers to the compiler/cpu) we do the same for second pixel in $v2$ and so on, and then if we add them together we get the output pixel values for 8 pixels.

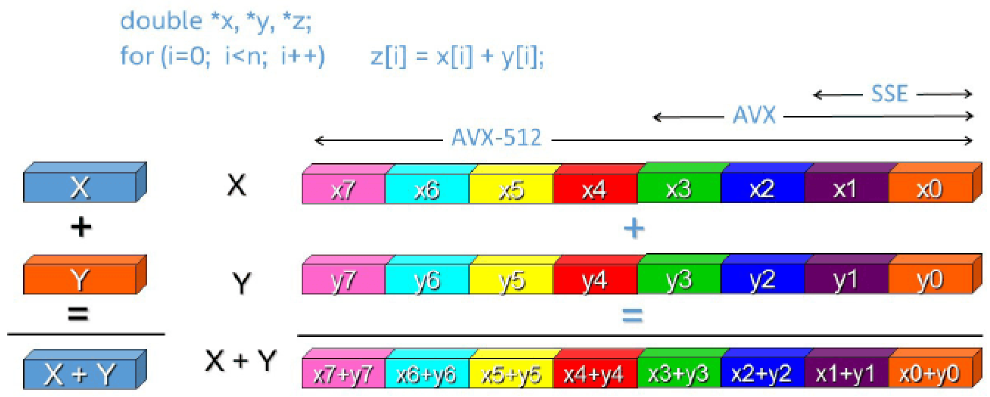


Figure 7: vector instructions

To load 8 floats into a 256 bit register from an address we do:

```
__m256 px1 = _mm256_load_ps(&img[idx_top]);
```

This loads 8 floats starting at *idx_top*, and the simplicity of this method is that the pixels are laid out sequentially in memory so for the first 8 output pixel, the first pixels are the first 8 pixel in the padded image, the second pixels are the 8 pixels with 1 shift, and so on. For the kernel, we need to fill a register with 8 k1s, another one with 8 k2s and so on, to do this we can use another intrinsic:

```
__m256 k1 = _mm256_set1_ps(kernel[0]);
```

Finally to store the results back in the output array, we use the store intinsic which puts back the 8 values in a register, as 8 elements in a float array given as first parameter, like this:

```
_mm256_storeu_ps(&result[8 * i], res);
```

This fills k1 with 8 of the float value in *kernel[0]*, so at this point all of the following code should be easy to understand (the other instructions used is add, and mul, which is trivial how they work).

```
void conv2d_1(const float* img, const float* kernel, float* result) {
        int idx_top = 0;
        int idx_res = 0;
```

```
__m256 k1 = _mm256_set1_ps(kernel[0]);
__m256 k2 = _mm256_set1_ps(kernel[1]);
__m256 k3 = _mm256_set1_ps(kernel[2]);
__m256 k4 = _mm256_set1_ps(kernel[3]);
__m256 k5 = _mm256_set1_ps(kernel[4]);
__m256 k6 = _mm256_set1_ps(kernel[5]);
__m256 k7 = _mm256_set1_ps(kernel[6]);
__m256 k8 = _mm256_set1_ps(kernel[7]);
__m256 k9 = _mm256_set1_ps(kernel[8]);
for (int i = 0; i < H; i++) {
        //_mm_prefetch((const char*)&img[(i + 1) * S], 4);
        for (int j = 0; j < W / 8; j++) {
                idx_top = i * S + j;
                idx_res = i * W + j;
                //__m256 v = _mm256_setzero_ps();
                __m256 px1 = _mm256_load_ps(&img[idx_top]);
                __m256 v1 = _mm256_mul_ps(px1, k1);
                __m256 px2 = _mm256_load_ps(&img[idx_top + 1]);
                //__m256 v2 = _mm256_fmadd_ps(px2, k2, v);
                __m256 v2 = _mm256_mul_ps(px2, k2);
                __m256 px3 = _mm256_load_ps(&img[idx_top + 2]);
                __m256 v3 = _mm256_mul_ps(px3, k3);
                __m256 px4 = _mm256_load_ps(&img[idx_top + S]);
                __m256 v4 = _mm256_mul_ps(px4, k4);
                __m256 px5 = _mm256_load_ps(&img[idx_top + S + 1]);
                __m256 v5 = _mm256_mul_ps(px5, k5);
                __m256 px6 = _mm256_load_ps(&img[idx_top + S + 2]);
                __m256 v6 = _mm256_mul_ps(px6, k6);
                __m256 px7 = _mm256_load_ps(&img[idx_top + 2 * S]);
                __m256 v7 = _mm256_mul_ps(px7, k7);
                __m256 px8 = _mm256_load_ps(&img[idx_top + 2 * S +
                ↪  1]);
                __m256 v8 = _mm256_mul_ps(px8, k8);
                __m256 px9 = _mm256_load_ps(&img[idx_top + 2 * S +
                ↪  2]);
                __m256 v9 = _mm256_mul_ps(px9, k9);
                /*      if (j == (W / 8 - 1)) {
                                 _mm_prefetch((const char*)&img[(i
↪  + 1) * S], 4);
                         }*/
                v1 = _mm256_add_ps(v1, v2);
                v8 = _mm256_add_ps(v8, v9);
```

```
                    v3 = _mm256_add_ps(v3, v4);
                    v5 = _mm256_add_ps(v5, v6);
                    v5 = _mm256_add_ps(v5, v7);
                    v5 = _mm256_add_ps(v5, v3);
                    v5 = _mm256_add_ps(v5, v8);
                    v1 = _mm256_add_ps(v1, v5);
                    _mm256_storeu_ps(&result[idx_res], v1);
                }
                //_mm_prefetch((const char*)&img[(i + 1) * S], 4);
        }
}
```

The result of the above implementation is remarkably fast, only 80 (us). I also made 2 other implementation using instrinsics which you can see in the complete source code, which implement the first idea of going pixel-by-pixel and use vector multiplication in the calculation of 1 pixel and then adding up the last (9th) pixel in. That one is slow due to the fact that vector instruction are meant to be used vertically: adding up the 8 float in 1 256 bit register (a horizontal calulation) is not fast, the third implementation is a slight variation of the second one, in that loading is done via another instruction (instead of *set ps*, I used *gather*), these 2 implementation ran at 500 (us) and 420 (us), still fast but not the best. In the resulting table, I named them such that Algo3 is the fastest and what we mostly discussed here with the source code. Also in the commented parts you see a *prefetch* function, which is an instruction which hints to the CPU that data at a certain address will soon be used, so that the CPU can preemptively bring it in cache so that later when we use it, we don't stall the core waiting for them to arrive in the cache, but they did not help much here.

2.6 CUDA

CUDA achieves parallelism in a different way than vector extensions, with CUDA unlike vector extensions where we were fixed to 8-way parallelism afforded to us, we can launch a CUDA kernel on multiple GPU threads which are typically much larger than just 8. A CUDA device function is the same code that will be run on all of them, and we have to mind which thread we are running on based on the index variables, these are complicated by the fact that threads are gathered in multidimensional blocks and grids. There are various references for it, but a good discussion is here. [7] Anyway, the CUDA kernel ran in roughly 60 (us) beating all other methods (no wonder). The code is basically the same as the first python code we had, but the indexing is now added so that each GPU thread calculates 1 output pixel.

Another issue with GPU, is that the GPU has a separate memory, so there is also an additional overhead, to copy data from RAM to GPU memory and back (using CUDA's memcpy). Integrated memory has no dedicated memory and shares the RAM with the

[7]https://stackoverflow.com/a/2392271/11216865

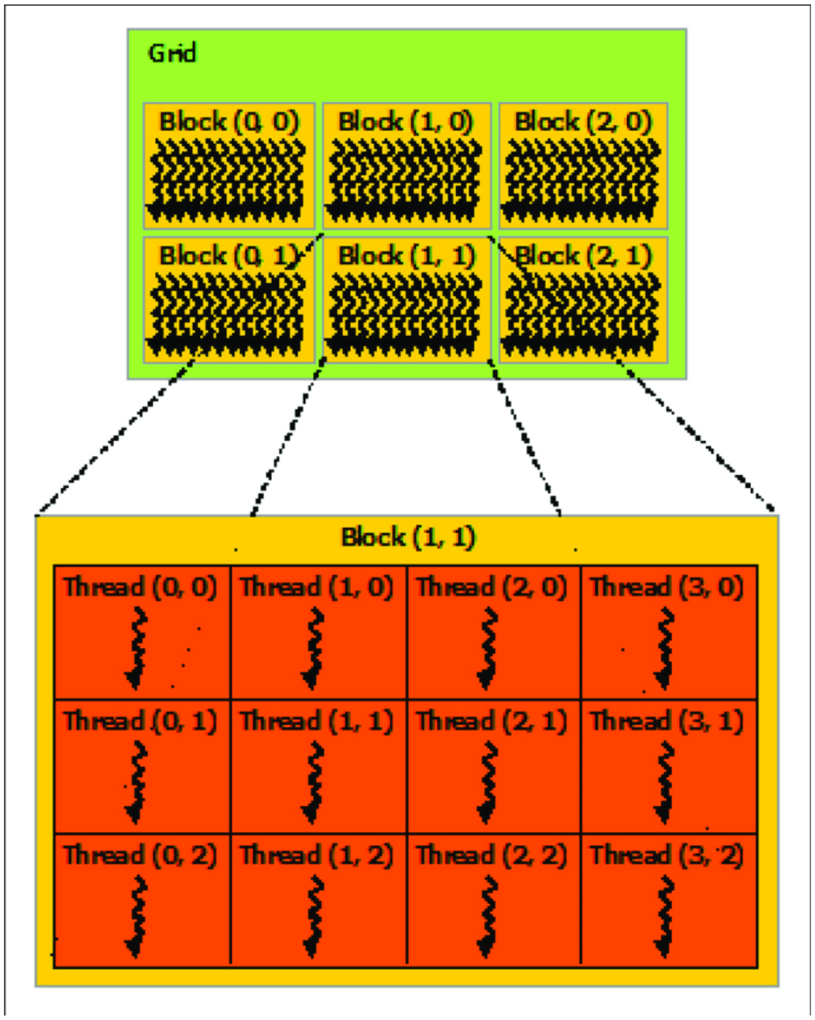


Figure 8: CUDA blocks and threads

CPU, and is worse off than the discrete GPU, because GPU memory is blazing fast (order of magnitude change) so once you get data to the GPU memory, accessing and changing it much faster.

2.7 Results

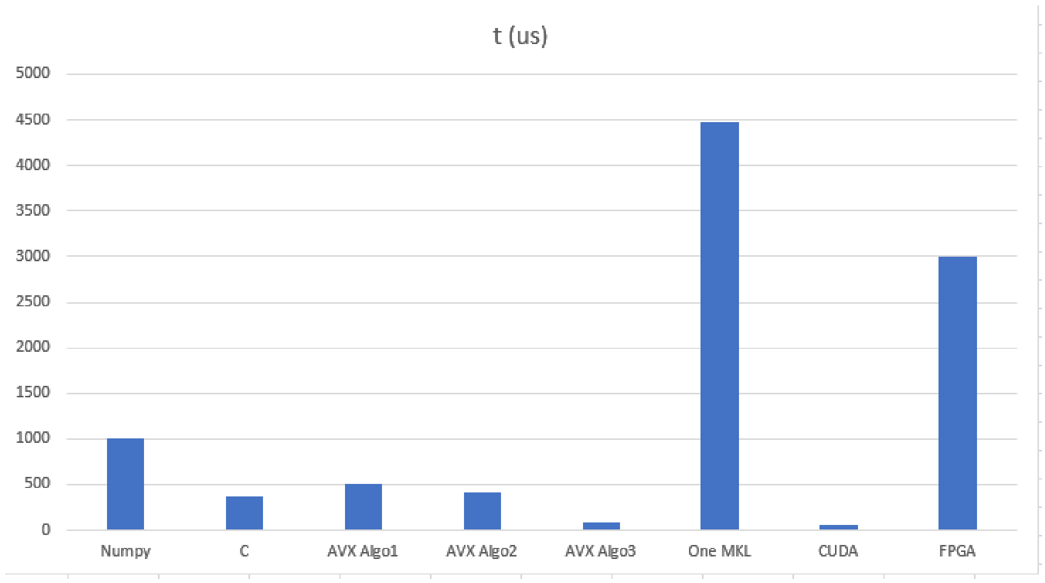


Figure 9: DELL XPS and FPGA

3 Galaxy Note S20

3.1 the setup

Galaxy Note S20 is powered with 8 heterogeneous cores, 2x Exynos M5, 2x Cortex-A76 and 4x Cortex-A55, also inculudes an ARM MALI-G77 GPU.

3.2 Implementation

In *MainActivity.java* you can see the java and OpenCV implementation (OpenCV implementation means basically just calling the OpenCV filter2D function, getting OpenCV to work was a pain though!). To use native code NDK (Native Development Kit) should be used which let's you compile your C code into a shared library (.so) file, and then use Java Native Interface (JNI) to call your native function from java (i.e. MainActivity). The native code has it's own CMakeLists.txt specifying it's compilation and in gradle files, it should be configured that you are using neon, and you want "mfloag-abi=hard" so that neon support works, because ARM cores can have a hardware floating point unit or they can do floating point ops in software. Parallelization was done by transforming our AVX code into NEON (which is easier than AVX) the main loop is:

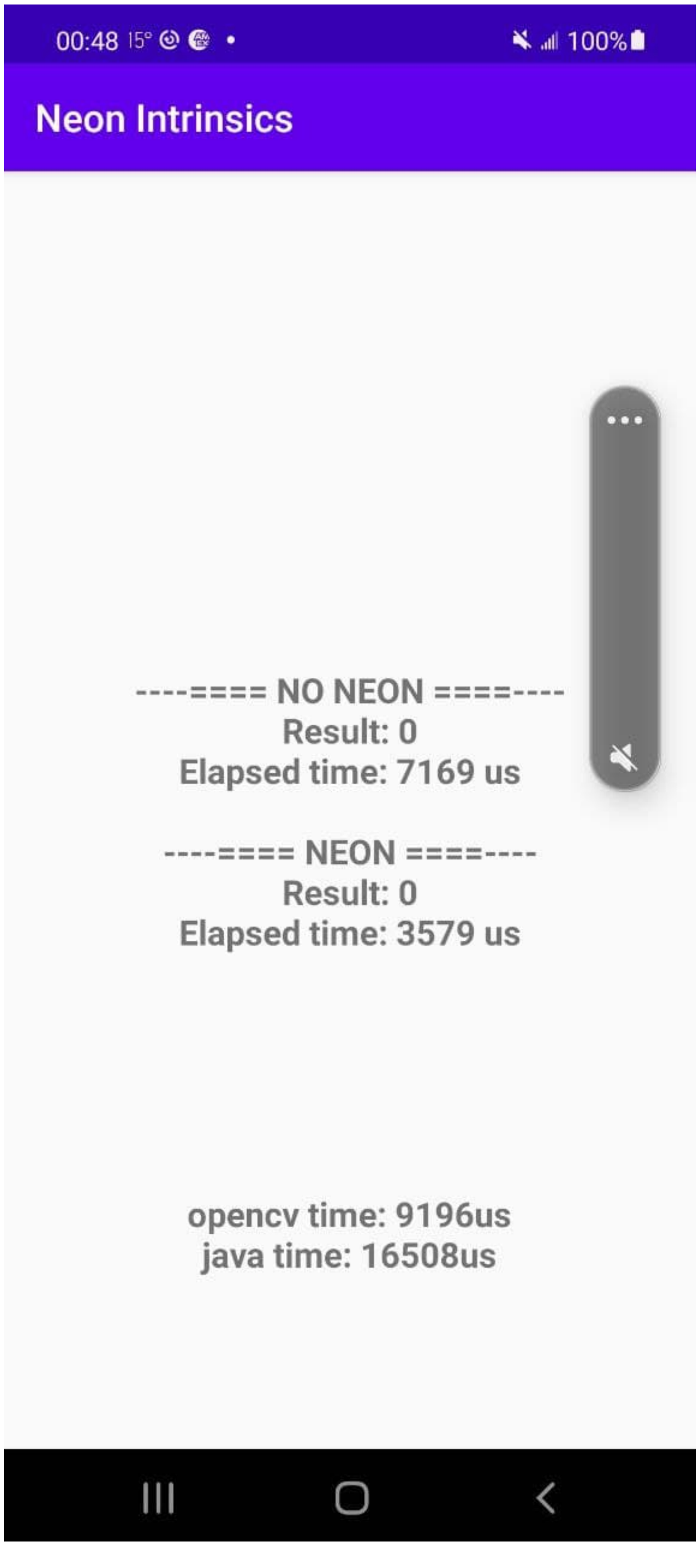
```
for(int i=0;i<width;i++) // i = rows
{
    for(int j=0;j<height;j++) // j = columns
    {
        for(int k=0;k<9;k+=3)
```

```
            {
                data[k] = imageData[i*width + j];
                data[k+1] = imageData[(i+1)*width + (j+1)];
                data[k+2] = imageData[(i+2)*width + (j+2)];
            }
            uint8x8_t pixel = vld1_u8((const unsigned char*)&data);
            uint8_t pixel_last = data[8];
            uint8x8_t result = vmul_u8(kernel,pixel);
            int sum = 0;
            for(int k=0;k<8;k++)
                sum += result[k];
            sum += pixel_last * kernel_last;
            sum = sum/16;
            sum = sum > 255 ? 255 : sum;
            newimageData[i*width + j]= uint8_t(sum);

        }
    }
```

Where you can see that unlike AVX instrinsics where we had those _mm256 laden unions and functions, here we have more intuitive *uint8_t* arrays and call *vld1_u8* to load them to a register and *vmul_u8* to multiply them element-wise, as you can see the result is directly in array form and we don't need a separate store operation like we did with AVX, this is why this code looks much more clean, though it ended up not being as fast (or achieve as much speedup).

3.3 Results



```
----==== NO NEON ====----
Result: 0
Elapsed time: 7169 us

----==== NEON ====----
Result: 0
Elapsed time: 3579 us

opencv time: 9196us
java time: 16508us
```

Figure 10: Galaxy Note running the code

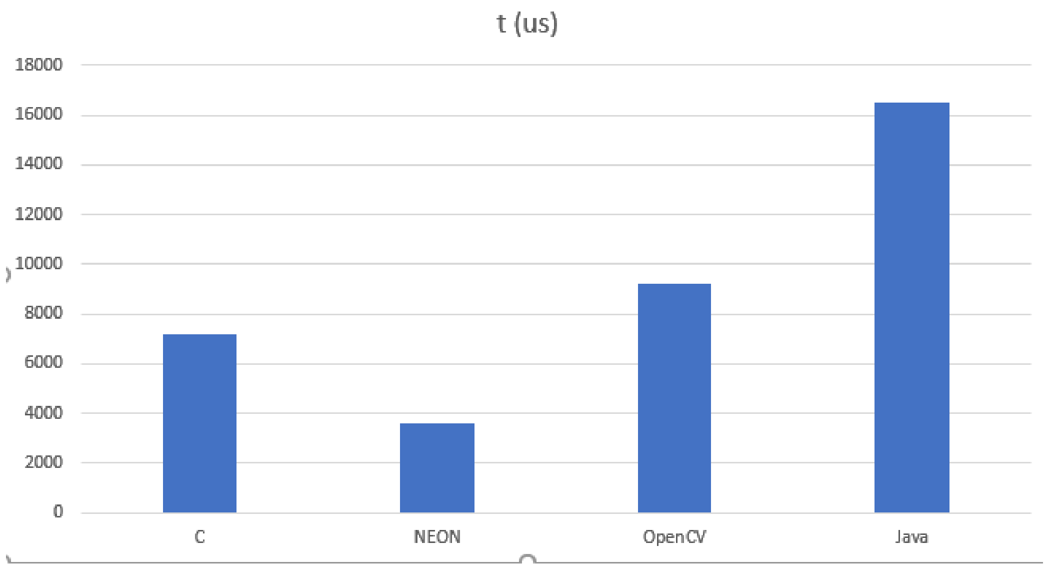


Figure 11: Galaxy Note

4 FPGA

The main gist of the FPGA implementation is summarized as:

```
always_comb begin
                sum_comb = 'd0;
                sum_comb = sum_comb + a1;
                sum_comb = sum_comb + a2*2;
                sum_comb = sum_comb + a3;
                sum_comb = sum_comb + a4*2;
                sum_comb = sum_comb + a5*4;
                sum_comb = sum_comb + a6*2;
                sum_comb = sum_comb + a7;
                sum_comb = sum_comb + a8*2;
                sum_comb = sum_comb + a9;
end

always @(posedge clk) begin
r <= (sum_comb >>4);
end
```

This simple block, generates logic circuit (in the top level, this is the dotter module) which does all these calculations *in parallel*, this is the most slick parallelism of all, at hardware level and at our discretion! At output, it does a right shift, to divide the values, here we do not have floats so to apply gaussian kernel we multiply by integer factors then divide the 16 factor to get the truncated results) It takes 1 clock cycle for this to happen (unlike how the Core i7 pipeline for example, can be as deep as 15 stages, which means more clocks) and another feature is that there is no longer speculation, no stalls and thus no non-determinism: FPGA performance can be very reliable. The UART needs a roughly 10 MHz clock (we are using 9600 baud rate - in the code it BAUD_RATE is half- baud rate) which we do via a clock divider, from the on board 50 MHz clock. The clock settings (and other pin settings) are set inside a constraints (.xdc) file which are based on the Digilent sample constraint file. The UART is a serial protocol where there is a single-bit line which is high on idle, and has 0 for stop and start, and in between it flips, sending in data in 8bits, then stop (or parity bits if set) and then repeat.

The input data (a 128x128 lena image) is written in *ram_init.coe* file, which is a file format telling Vivado to fill the block RAM with that inital data. Then the addr signals to the RAM are set and data is read to 9 registers for the 9 pixels under consideration (and set to 0 signal for out of range ones, which does the zero-padding) and then sent to the dotter module which applies the filter and loads output with the result pixel, which is loaded into an output RAM. When all the data is ready, it is transmitted over UART.

Without doubt, the FPGA implementation was the hardest, and was the first one I started working on.

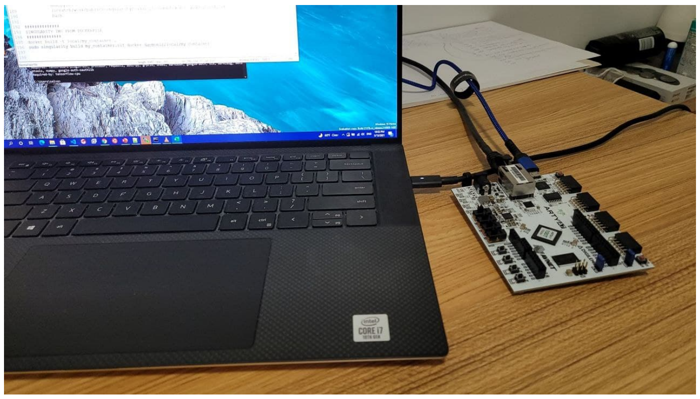


Figure 12: FPGA connected to the laptop

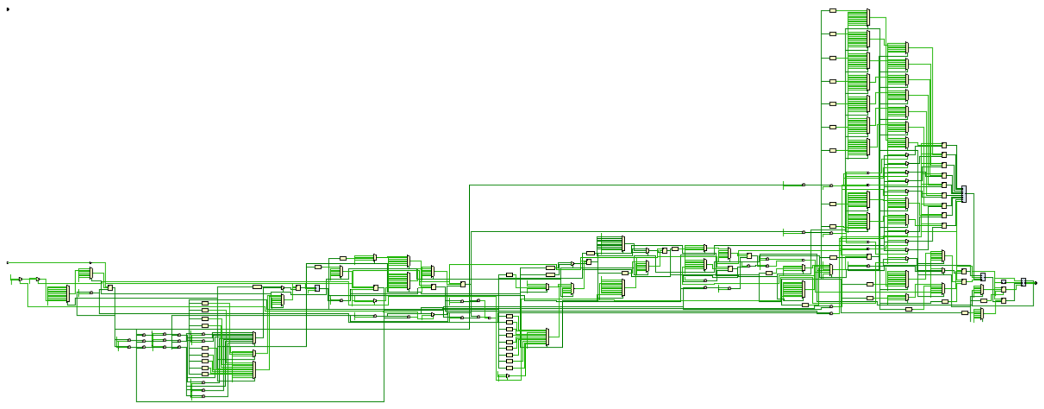


Figure 13: Top level schematic is not very helpful

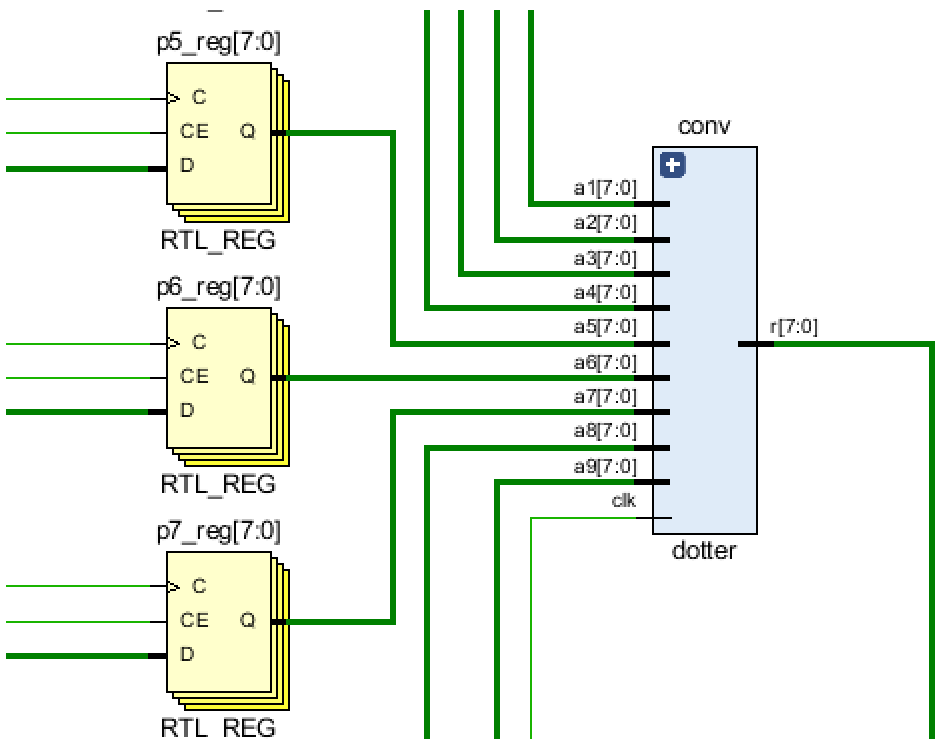


Figure 14: Zoomed in to the conv module