

19:10 Vector Multiplication as an IPC Primitive

by Lorenzo Benelli

Since time immemorial computer scientists have pondered what could be the best way for two processes to interact with each other. Is it shared memory? Is it message queues? Is it sockets? Wait no more, dear neighbor, because in this modest article I'm going to present a novel and more promising way. We will see that processes can communicate with one another by using little more than vector instructions!

Overview of power management

Starting with the Sandy Bridge architecture, Intel's ISA included a new set of instructions called AVX, to operate on larger, 256-bit sized, registers. More recent architectures further extended this functionality with another set, AVX2.

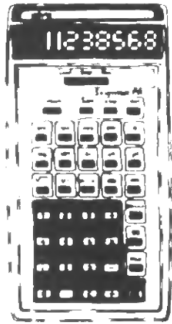
As keeping these wide registers turned on all the time wasn't power-efficient, Skylake and later architectures kept them inactive during the normal scalar code execution. The CPU would start powering on these wider, vector data paths only when the first SIMD instruction got executed.

This process takes time, and while the vector execution units are being turned on, the vector code gets dispatched to μ ops that make use of narrower registers and, consequently, execute at roughly half the speed. Also, after the core encounters a vector instruction, the processor will keep the registers active for a while (on the order of milliseconds) after the last SIMD instruction is scheduled to run.

As the core that runs this sort of vector code will require more power to keep the registers active, the Package Control Unit (PCU)—an on-chip microcontroller that manages frequencies and voltages of the processor—will increase that core's voltage with a mechanism that Intel calls "granting a power license."

Within the bureaucratic apparatus that is the processor, a core is granted a different power license depending on the kind of instructions it is executing. For all AVX instructions, and for some simple AVX2 instructions like loads and adds, the core gets to run on the modest LVL0_TURBO_LICENSE. For complex AVX2 instructions it gets the regular LVL1_TURBO_LICENSE, while the cores lucky enough to run AVX-512 win a premium LVL2_TURBO_LICENSE.

PROGRAMMABLE
Scientist
\$29⁹⁵



100 STEPS
100 STEP LEARN
MODE, KEYBOARD
PROGRAMMING
CAPABILITY.

- RPN logic • Rollable
- 4-level stack • 8-
- digit plus 2-digit exponent LED display
- Scientific notation • Sine, cosine,
- tangent & inverse trigonometric functions
- Common & natural logarithms & anti-
- logarithms • Instant automatic calcula-
- tion of powers and roots • Single-key
- square root calculations • Single-key Pi
- entry • Separate storage memory •
- Square, square root and reciprocal cal-
- culations • Change sign & register ex-
- change keys • Includes NiCad batteries.

Mfg. by National Semiconductor
1 year warranty • 10 day money back guarantee

100 pg. Application Handbook - \$5.00;
AC charger - \$4.95;
Carrying case - \$2.95; Stand - \$2.00;
Ship & hndl. \$3.75.

TO ORDER CALL
(213) 559-1044
OR SEND CHECK TO
ILDAN Inc.
6020 Washington Blvd.
Culver City, CA 90230

12

Also, the core's frequency gets capped by the PCU to a lower value, which is referred as the AVX2 Turbo frequency. For commercial desktop and laptops CPUs, this applies to not just the core running vector code but to all cores in the same processor.

This led me to wonder: what is happening to the wide SIMD units of the other cores during that time? Are they all powered-on all together? If so, could this be used to make our processes have a little chat without bothering the OS with expensive syscalls?

Latency is key

With this rough idea of the inner workings of the Intel's CPU power management, I wrote a tiny snippet of code that launches two processes with the ability to communicate without any nasty interaction with the OS.

```
1 #include <immintrin.h>
2 #include <stdio.h>
3
4 #define TIME_SCALE 1.0
5 #define BUFSZ 0x400
6
7 void bsleep(uint64_t);
8 void send(uint8_t);
9 void recv(void);
10
11 int main() {
12     pid_t pid;
13
14     if ((pid = fork()) == 0) {
15         recv();
16     } else if (pid != -1) {
17         send('P');
18         send('o');
19         send('C');
20         bsleep(0x40000000);
21         kill(pid, 9);
22     }
23     return 0;
24 }
25
26 void bsleep(uint64_t clk) {
27     uint64_t beg, end;
28     uint32_t hi0, lo0, hi1, lo1;
29     asm volatile (
30         "cpuid\n\t"
31         "rdtsc\n\t"
32         "mov %%edx, %0\n\t"
33         "mov %%eax, %1\n\t"
34         : "=r" (hi0), "=r" (lo0) ::
35         "%rax", "%rbx", "%rcx", "%rdx"
36     );
37     end = beg = (((uint64_t)hi0 << 32) | lo0);
38     while (end - beg < clk) {
39         asm volatile (
40             "cpuid\n\t"
41             "rdtsc\n\t"
42             "mov %%edx, %0\n\t"
43             "mov %%eax, %1\n\t"
44             "pause\n\t"
45             : "=r" (hi1), "=r" (lo1) ::
46             "%rax", "%rbx", "%rcx", "%rdx"
47         );
48         end = (((uint64_t)hi1 << 32) | lo1);
49     }
50 }
```

SAM COUPE AND SPECTRUM MAGAZINE!
PROGRAMS, UTILITIES, INFO, IDEAS! NEWS, REVIEWS AND HOMEGROWN SOFTWARE MONTHLY SINCE 1987!
GRAPHICS, AND HELP PAGES, SERIOUS SOFTWARE
"OUTLET"
SPECIAL OFFER! Latest issue £2.50 to newcomers on:-
+3, DISCIPLE/+D, MICRODRIVE, OPUS, TAPE, SAM DISC
CHEZRON SOFTWARE, 605 LOUGHBOROUGH Rd., BIRSTALL, LEICESTER LE4 4NJ

One parameter offered by the code is TIME_SCALE, which you can set at your convenience in case your plotting utility doesn't implement horizontal zooming, or if you wish to pin the processes to far away cores.

As we'd like to eventually store some measurements, BUFSZ provides a way to delay the unavoidable write() call, because the longer we can prolong our abstinence from kernel communication, the better.

For each bit to be transmitted, the sender process either executes a *very long* succession of AVX2 multiplications, or enters a busy loop, doing nothing for long enough that the PCU decides to revoke its power license, powering off the vector execution units.

Another process, the receiver, runs a *short* burst of vector instructions, then also sleeps for enough time that the PCU decides to revoke its power license. The receiver process is also keeping track of its execution speed via the rdtsc instruction, periodically dumping it to stdout.

```
1 void send(uint8_t c) {
2     for (int i=0; i<8; i++) {
3         uint8_t bit = (c >> i & 1);
4         if (bit) {
5             for (uint64_t i=0; i<0x4000*SCALE; i++){
6                 asm volatile(
7                     "pushq $0x40000000\n\t"
8                     "vbroadcastss 0(%%rsp), %%ymm0\n\t"
9                     "vbroadcastss 0(%%rsp), %%ymm1\n\t"
10                    "mov $10000, %%ecx\n\t"
11                    "loop1:\n\t"
12                    "vmulps %%ymm0, %%ymm1, %%ymm1\n\t"
13                    "dec %%ecx\n\t"
14                    "jnz loop1\n\t"
15                    "popq %%rcx\n\t"
16                    :::
17                );
18                bsleep(0x20000);
19            }
20        } else {
21            bsleep(0x8db6db6d * SCALE);
22        }
23        fprintf(stderr, "tick %d\n", bit);
24    }
25 }
```

```

1 void recv(void) {
2     uint64_t beg, end, i = 0;
3     uint32_t hi0, lo0, hi1, lo1;
4     static uint64_t time[BUFSZ];
5     static char buf[0x10000], *it = buf;
6
7     while (1) {
8         asm volatile (
9             "cpuid\n\t"
10            "rdtsc\n\t"
11            "mov %%edx, %0\n\t"
12            "mov %%eax, %1\n\t"
13            : "=r" (hi0), "=r" (lo0) ::
14            "%rax", "%rbx", "%rcx", "%rdx"
15        );
16        asm volatile(
17            "pushq $0x40000000\r\n"
18            "vbroadcastss 0(%%rsp), %%ymm0\r\n"
19            "vbroadcastss 0(%%rsp), %%ymm1\r\n"
20            "mov $10000, %%ecx\r\n"
21            "loop:\r\n"
22            "vmulps %%ymm0, %%ymm1, %%ymm1\r\n"
23            "dec %%ecx\r\n"
24            "jnz loop\r\n"
25            "popq %%rcx\r\n"
26            :::
27        );
28        asm volatile (
29            "cpuid\n\t"
30            "rdtsc\n\t"
31            "mov %%edx, %0\n\t"
32            "mov %%eax, %1\n\t"
33            : "=r" (hi1), "=r" (lo1) ::
34            "%rax", "%rbx", "%rcx", "%rdx"
35        );
36        beg = (((uint64_t)hi0 << 32) | lo0);
37        end = (((uint64_t)hi1 << 32) | lo1);
38        time[i++] = end - beg;
39
40        bsleep(0x1000000);
41
42        if (i == BUFSZ) {
43            i = 0;
44            for (uint64_t i = 0; i < 1024; i++) {
45                it += sprintf(it, "%lu\n", time[i]);
46            }
47            printf("%s", buf);
48            it = buf;
49        }
50    }
51 }

```

If the receiver process is running during a quiescent period of the sender process, meaning that the vector registers are powered down, it will run at about half the speed for at least 150K clock cycles, which is roughly the warm-up period on Coffee Lake. Otherwise, it will dash forth at full speed. Repeating this enough times, the receiver can gather sufficient evidence to know what bit was being sent to him by his neighboring process.

On page 58 you can see the data plots taken from some Kaby, Coffee Lake, and Sky Lake systems, and a reference of the inverted ASCII signal, where the most significant bits are sent last.

The End

What is actually happening inside the processor is not completely clear to me. Perhaps the vector units are not kept active *all* the time while executing AVX code. Since the PCU on mixed scalar/vector workloads has already lowered the frequency of all the cores, it has more room to adjust their voltages quickly, and it is consequently able to power the wide paths faster, ultimately with similar effects. Let me know if you manage to figure this out, neighbors!

Finally, a few words about why I think this is a better way for processes to communicate.

First, the processes get to avoid those pesky `syscall` instructions which make the software we write daily completely non-portable.

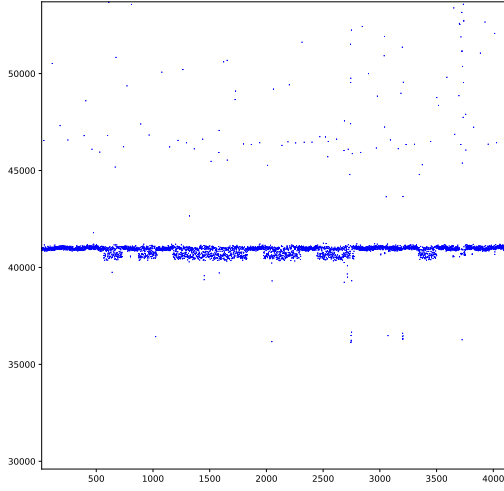
Second, although not as fast as other IPC implementations, this one makes communication a CPU-bound problem instead of an I/O-bound one, which, as everybody knows, is a much nicer problem to have.

Third, two processes in completely separate VMs can now communicate, without the extra long and boring configuration jobs that sysadmins have to do in order to get the infrastructure to work.

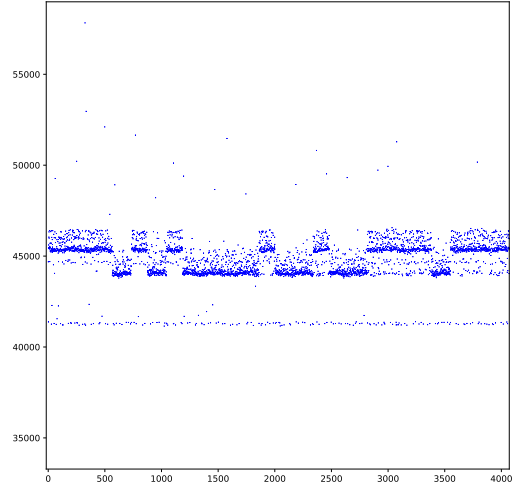
This is why, neighbors, you should promptly experiment with this method, as well as try to find further novel and nifty ways to use our processors. Maybe we will one day be able to multiply two vectors with only `syscall` instructions!

**Employees must
wash hands before
returning to libc**

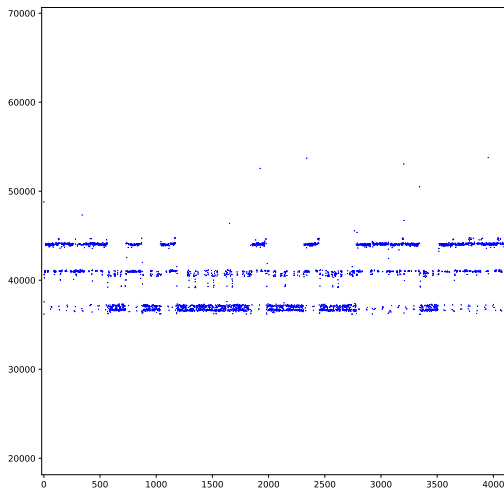




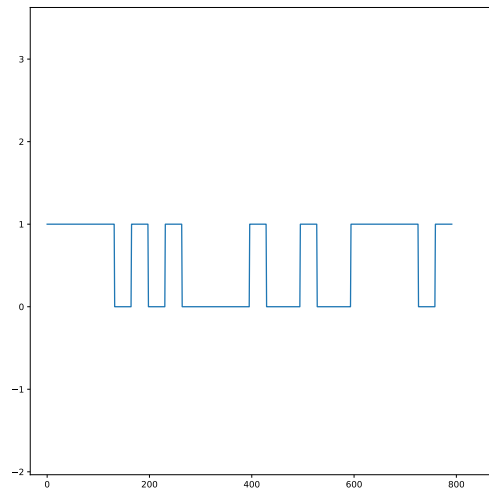
Coffee Lake Warmup Time



Sky Lake Warmup Time



Kaby Lake Warmup Time



Reference Message (POC)