Power aware efficient programming · universidad-zaragoza/designing-modern-processors Wiki

Hands-on Lab on Energy/Power Aware-Programming

Preliminaries:

To run the experiments, we will use the socarrat.unizar.es machine that belongs to the Computer Architecture Group at the University of Zaragoza. Your user id is the same for all the machines in the DIIS, *a letter followed by your NIP*, and the password is `centro6-datos2-computadores8`.

Measuring Energy and Power on linux

To measure total power (dynamic + static), we are going to rely on the Intel power hardware counters, RAPL, of an Intel Skylake processor (4 cores, 2 threads/core) and read them with the `perf` tool. Since these counters are global and not per application, the results may be incorrect with multiple users. You can run the tool with:

```
$ perf stat -a -e "power/energy-cores/" <cmd>
```

Also, before continuing with the lab, please take a quick look to the man page of perf command and manual to learn the the basics of the tool.

As said, since the counters are not replicated for every core, if several of you run an experiment at the same time, the measured energy will correspond to the sum of the energies of all the running experiments. To get accurate results, please share a token on a chat. Also, you can perform the lab in couples to reduce the total number of runs.

Intel RAPL offers readings from multiple power domains. The `power/energy-cores/` domain measures only the energy of the cores. The `power/energy-pkg/` returns the energy consumption of the whole package; this domain manages the entire CPU socket, which includes all the processor cores as well as the uncore components (e.g., the last-level cache, integrated GPU, and memory controller). Finally, the `power/energy-ram/` reports the DRAM energy consumption.

Building the examples

Once you are able to measure energy and power and to compile code, it is time to clone the repo and build the example programs. To do so, please login again with your account into socarrat and run the following commands:

```
git clone https://github.com/universidad-zaragoza/designing-modern-processors.git

cd designing-modern-processors

mkdir build-release

cd build-release

cmake -DCMAKE_BUILD_TYPE=Release ../

make -j$(nproc)
```

The repo uses cmake as build system that encourages to keep your binary files separated from the source code.

Now, you are ready to start analyzing how different aspects affects the energy and consumption of your code.

Effect of Parallelism in Performance and Energy consumption on CPU

The first experiment of the lab analyzes a simple C++ code that approximates π with a Taylor series.

The π number represents the ratio of a circle's circumference to its diameter, and its value was already approximated before the Common Era. The ancient Greek Archimedes already devised a π approximation algorithm.

With a Taylor series, π can be approximated as

$$\pi = 4 \sum_{i=0}^{\infty} \frac{-1^n}{2n+1}$$

The C++ source code of the program is located in the pi-taylor directory of this repository and is named pi_taylor_parallel.cc.

Within the source code, the `pi_taylor_chunk` functions computes and accumulates a chunk of steps from the approximation, so multiple threads can share the work because the steps are independent among them.

```
void
pi_taylor_chunk(std::vector<my_float> &output,
        size_t thread_id, size_t start_step, size_t stop_step)
{

    my_float acc = 0.0f;
    int sign = start_step & 0x1 ? -1 : 1;

    for (size_t i = start_step; i < stop_step; ++i) {
        my_float divider = 2.0f * i + 1.0f;
        acc += static_cast<my_float>(sign) / divider;
        sign = -sign;
    }
    output[thread_id] = acc;
}
```

Then, the main program launches a number of threads, waits for their completion, and performs the final addition and division to estimate π.

```
    thread_vector.reserve(threads);
    for(size_t i = 0; i < threads; ++i) {
        auto begin = i * steps_per_thread;
        auto end = std::min(begin + steps_per_thread, steps);
        thread_vector.push_back(std::thread(save_times, begin, end, i));
    }

    for(auto &t: thread_vector) {
        t.join();
    }

    my_float pi = 4.0L * std::accumulate(partial_results.begin(),
        partial_results.end(), 0.0L);
```

Please do not hesitate to ask about the code if you have any question. Inside your `build-release/pi-taylor` directory, there should be a `pi_taylor_parallel` binary. Two arguments control its execution, the number of steps and the number of threads. During the lab, the number of steps can be set to 100000000 to run the program just long enough.

First, please run the program with perf (the -r 5 argument repeats the experiment 5 times) and a single thread.

```
$ perf stat -a -e power/energy-cores/ -r 5 ./pi_taylor_parallel 100000000 1
```

Then, run the program with 2, 4, 8, and 16 threads, and plot the results with the number of threads in the X axe and the execution time and total energy in the Y axe. You can use an online service such as colab, jupyter, your favorite spreadsheet, or your preferred tool for plotting.

Questions

1. Do total energy and execution time correlate? Why?
2. Does average power increases when the number of threads does so? Why?
3. Is the total amount of work done very similar between runs with different number of threads? If yes, why energy may depend on the number of threads ?
4. Based on the responses to the previous questions, do you think that the control parts of the processors such as clock distribution, instruction fetch, decode and control can represent an important part of the energy consumed?

Effect of locality on power and energy in Matrix Multiplication on CPU

One key programming optimization to save energy is locality. In general, when a program reuses data, there are fewer data movements between the memory hierarchy and the execution units, saving energy and reducing execution time. Matrix Multiplication is a problem where locality plays a significant role. The access pattern of each input matrix is opposite, while one matrix is traversed by rows, the other one is traversed by columns. Assuming a row-major order as C++ does, traversing by rows will ensure good spatial locality, while the stride access of the column-based traversing will probably trash the cache and waste a lot of energy bringing unused data to the on-chip caches.

To check the effect of locality, in this exercise, you will compare a naive matrix multiplication implementation with a highly optimized version from the Eigen library [1]. The code of both versions is located inside the matrix-multiplication directory. Please check both versions and try to make a mental model of the cache accesses to assess their locality (both spatial and temporal).

Questions

1. Please run both versions and compare execution time, average power, and total energy. Are the energy gains for the package equal to the core gains?

1: *Besides using tiling, transposing matrices, ... Eigen also can vectorize the code, so the comparison is not entirely fair.*

Effect of device selection on performance and power

Continuing with the matrix multiplication example, the next step consist on comparing the Eigen optimized CPU version with their GPU counterpart based on the cuBLAS library. This library provides a highly optimized version of Basic Linear Algebra Subprograms, BLAS, on GPU. On the socarrat system, the GPU is a Nvidia GeForce RTX 3090 Ti. To estimate the GPU power consumption, Nvidia provides the `nvidia-smi` tool that queries the GPU and prints the power drawn, among other things. In our particular case, the following command prints the power drawn every second during ten seconds:

```
$ nvidia-smi dmon -s p -c 10
```

Since the execution time of the code and the period of the tool is large, to estimate the power and energy, we are going to assume that the maximum value read represents the average power for the whole execution. For example, for the following output, the assumed average power will be 96 W.

```
[dariosg@socarrat man]$ nvidia-smi dmon -s p -c 10
# gpu   pwr gtemp mtemp
# Idx     W     C     C
    0    15    51     -
    0    15    51     -
    0    15    51     -
    0    15    51     -
    0    62    53     -
    0    95    53     -
    0    96    53     -
    0    96    52     -
    0    60    53     -
    0    95    54     -
```

Now, please run the eigen_matrix_multiplication, CPU, and cublas_matrix_multiplication, GPU, applications and collect the execution time and energy of both executions for matrices of the same size.

Questions

1. Open the cublas_matrix_multiplication.cc file and find where the time measurement process starts and stops. Does the measured time includes the data transfer time between CPU and GPU or only the execution time on the GPU? Why?
2. How many FLOPS operations perform the program, only in the matrix multiplication? Please compute the energy efficiency (in terms of FLOPS/Jules) of the CPU and the GPU? Which device is more efficient if we only account for the device themselves?
3. While the GPU performs the matrix multiplication, the CPU is also active? What is the energy efficiency when the host CPU energy is included?
4. Based on the previous results? Are GPUs a good accelerator for heavy computing tasks?

Effect of Load Balancing on power and energy

In the fork-join parallel pattern, load unbalance occurs when each thread has a different amount of work, and unbalanced parallel code tends to perform poorly. Sometimes, the inner nature of the algorithm enforces the unbalance, making very hard to ensure that all threads finish their work at the same time.

To *simulate* the effect of unbalance, we are going to modify the parallel pi_taylor code. In the provided version, all the threads performs the same amount of work that is the ratio between the number of steps and the number of threads.

Please create a new pi_taylor_unbalanced based on the pi_taylor_parallel.cc source code. Modify the program to run always with for 4 threads, 100000000 steps, and add 4 arguments representing the amount of work that each thread should perform. All these arguments together should add 1.0.

Then run the following combinations of load: [0.85, 0.05, 0.05, 0.05], [0.3, 0.3, 0.3, 0.1] and [0.5, 0.2, 0.15, 0.15], plot the results for execution time and total power with the combinations on the X axe, and answer the following questions;

Questions

1. Does the unbalance affects the execution time and the total energy of the program?
2. Does the unbalance affects average power?
3. Would a dynamic balancing policy help?

Impact of idle time in energy

Until now, all experiments execute a program, and the energy consumption corresponds to "useful work". However, what happens when the system is idle doing *nothing*? Without any support, the system would be consuming useless energy, and modern processors implement the ACPI power states to sleep, and reduce consumption, when idle. ACPI distinguish several power states, starting from C0 and going to deeper sleep states that saves more energy C_x_. On the Intel Nehalem microarchitecture of `socarrat`, we have the following states:

1. *C0*: Active state, the processor is not executing instructions
2. *C1*: Idle state, the processor executes the `HALT` instructions, so the clock stops.
3. *C1E*: Idle state, enhanced C1 state when Vdd also reduces
4. *C2, ..., C6: Idle states, where the processors go to a deeper sleep state as the state number grows.

For Intel cpus in the linux kernel, the cpuidle driver manages idle time and provides statistics on how long each logical CPU has been in each state.

Questions

1. Please pick a single logical CPU, from 0 to 7, and enter the `/sys/devices/system/cpu/cpu<N>/cpuidle/` directory. There perform an `ls` and with the cpuidle manual describe the information contained at the following files: `above`, `bellow`, `residency`, `latency`, `time`, and `usage`.
2. Since the last boot of the machine, what is the percentage of time that your selected CPU has been in the C6 state? *The command `uptime -p` returns how long the system has been running*
3. Assuming this CPU experiences the following conditions: 1) staying at C6 reduces power by 85% of the Thermal Design Power (TPD), and 2) the TPD of the processor is 130 W, see red hat documentation, how much energy cpuidle has saved compared to having remained at C0 state at half of the TPD for two hours?

Optional Effect of frequency in performance and energy efficiency

Modern processor provide support to adjust their power consumption with the adjustment of processor frequencies, aka Dynamic Voltage Frequency Scaling, via the ACPI P-states. In order to measure the impact of voltage-frequency scaling on performance, we are going to fix the application, Eigen's based parallel matrix multiplication on CPU, and then tune the processor frequency to see what are the differences in execution time and energy. Note that in `socarrat.unizar.es` the frequencies of all cores work in lock-step mode, so all frequencies are always the same.

In order to perform this optional section, we have to enable a power governor that allows to adjust the processor frequency from userspace. In Linux there are several options. First, the `userspace` governor provides this capability and can be enabled with:

```
$ echo "userspace" | /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor # with super-privileges
```

Then it is possible to adjust the frequency of the cores through the same interface that is explained in the Linux documentation. Second, also, on some Intel CPUs, the intel_pstate driver enables changing the frequency as well. Currently, socarrat.unizar.es uses the `intel_pstate` driver, so the frequency adjustments will be performed by tweaking the available frequency ranges of the governor with two attributes: `min_perf_pct` and `max_perf_pct` located inside the `/sys/devices/system/cpu/cpufreq/intel_pstate/` directory. To simplify the task, change_frequencies_p_states.sh adjust the attributes to the desired values and measures the energy consumption with `perf`.

Questions

1. First ensure that matrices are 8192 by 8192, otherwise the execution time will be very small. You may have to modify the source code for that. Second, please run Eigen's parallel version on the CPU and the lowest (19, 19) , highest (100, 100), and middle frequencies (60, 60) with all the cores enabled with the provided script. Then compute the FLOPs/Jule and discuss the results. Please note that `intel_pstate` uses values between 0 and 100 to represent the frequencies. To know the running frequency of the cores please use the `sudo cpupower monitor -m Mperf` command that displays the exact frequencies as reported by the kernel. There will be some minor differences in KiloHertz, so for the sake of simplicity, please round to the next hundredth value. For our particular cases, the lowest, middle, and highest frequency corresponds to 800, 2600, and 4200 MHz, respectively.

Evaluation

Fill in the responses in a brief report, 10 pages maximum, with your answers and submit it through moodle as a pdf file. *Please do not include any screen capture unless you justify its value.*