Performance Lab

Notes

- **Due date: Nov 18th 2022, 11:55pm PST**
- This is an individual assignment – all code must be your own – no code sharing.
  It is fine to discuss optimizations with another student, but if you do, please write their name as a collaborator at the top of kernels.c.
- We made a bunch of changes to this lab this year … apologies in advance for any problems.
- Lab download / scoreboard website
- Lab TA: Salekh – Thanks Salekh for writing the scoreboard for this lab! Please send all your bug fixes to him!

Introduction

In this lab, your job is to optimize two simple “kernels” to make them faster and more energy efficient!

More specifically, given a suboptimal implementation of two kernels below, your goal is to utilize the optimization techniques you learned from class to speed up the calculation and improve “energy efficiency”. In this class, we approximate energy efficiency as the reduction in data movement across different levels of the memory hierarchy (i.e. so you should try to hit in the L1 cache as much as possible).

You’ll have to use some optimization techniques covered in the class, especially the cache and memory optimization ones.

Kernel 1: 2D Transpose

In 2D transpose, we take a 2D input matrix (Ni by Nj), and swap its rows and columns to create a 2D output matrix (Nj by Ni).

We can define the transpose mathematically as follows:

$$Out_{i,j} = In_{j,i}$$

Outi,j=Inj,i

Hints specific to 2D Transpose

- Think about the cache access pattern – it seems like we get bad spatial locality on either the In or Out array.
- Can we use tiling to help?

- I don't think this version will get vectorized by GCC, so don't worry if vector instructions are not showing up.

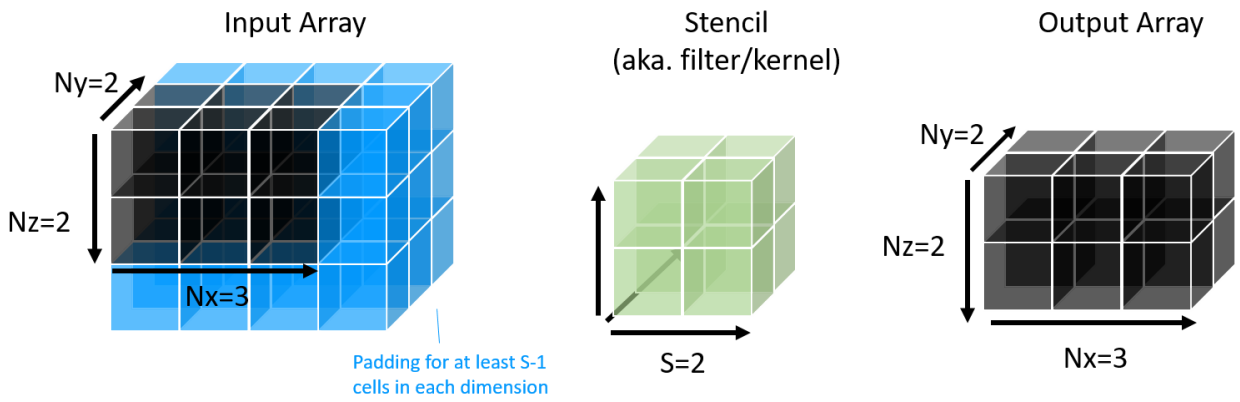
Kernel 2: 3D Stencil

Stencil algorithms "apply" a stencil (sometimes called a filter) to an input array to produce an output array. Stencil methods are quite common in image processing and computer simulations.

This stencil algorithm involves 3 arrays:

- "Input": This is a large 3D array of input floating point values, with dimension sizes Ni, Nj, Nk. It has some padding in each dimension, based on the size of the Stencil array.
- "Stencil": This is a smaller array of floating point values, with size S in all dimensions, which gets shifted over the input.
- "Output": This is the same size as the input array, but without padding.

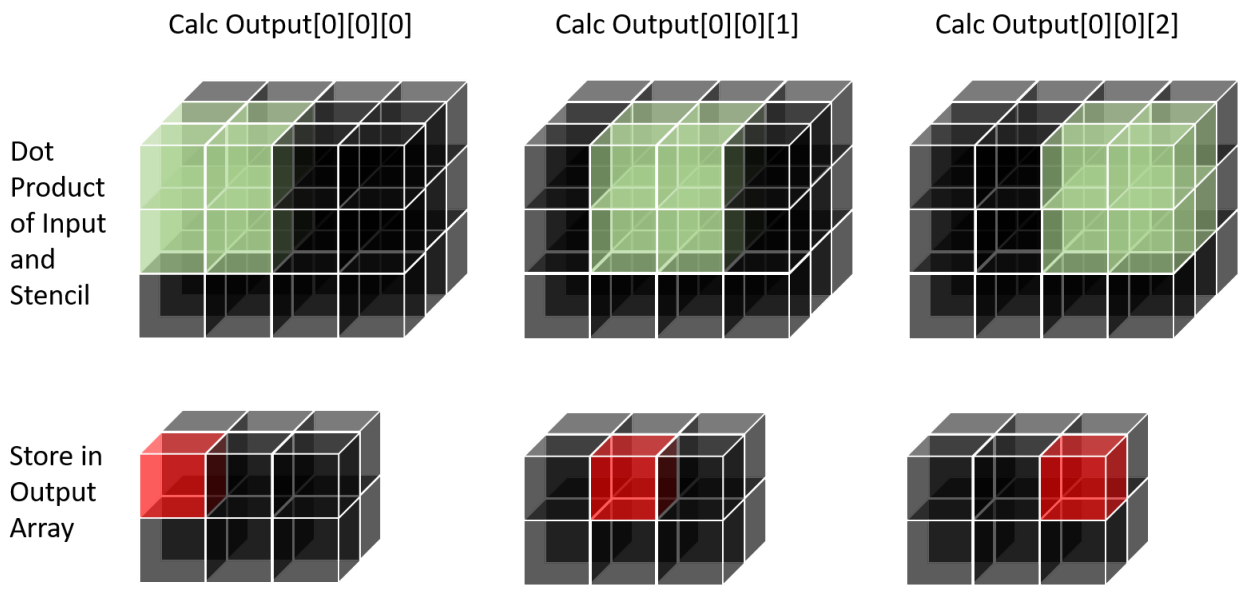
Below is a visualization of the three arrays involved:



Intuitively, the stencil algorithms shift the stencil around to every position on the input grid. For each position, we perform a dot product of the cells of Stencil and Input, and this result is written in the corresponding position of the Output array.

In this lab, we'll define the 3D stencil for each cell of the output as follows:

Below we depict the computation process required for the first three pixels along the x dimension.



Note that there is significant data-reuse between different computations. The stencil values are reused completely at each iteration.

Hints specific to 3D Stencil

- There is so much reuse that tiling is not so important here. (unless you really want to get extra credit, then maybe…)
- The loop ordering matters a lot, though. We really need to order the loops to get good temporal and spatial locality.
    - It will help you play with the loop ordering if you zero-out the Output array in a separate loop.
- Create a version of the code specifically for each problem size (Ni,Nj…). When more of the variables are constants, the compiler can generate more efficient code.
- GCC is definitely capable of vectorizing this code. Make sure that the statistics indicate that you are executing vector instructions – if not, your inner loop might not be chosen well to get contiguous access.

Scoring your Performance / Energy

For performance, we measure execution time. For energy efficiency, we are approximating it with “memory energy” for simplicity. Specifically, we use run your program with performance counters, and then multiply each access to L1/L2/L3 and DRAM by a somewhat arbitrary constant. The memory energy is defined as:

Mem Energy = #L1_Accesses * 0.05nJ + #L2_Accesses * 0.1J + #L3_Accesses * .5nJ + #DRAM_Accesses * 10.0nJ

To get your final score, we will run your code with 4 different sets of matrix sizes. Test case 1 is for transpose, and test cases 2-4 are for stencil. To combine your performance and energy scores into a single metric, we first take the geometric mean of your performance and energy improvement scores. Then we compute “energy delay improvement”, which multiplies your geometric mean performance and energy benefit together:

$$energy\_delay\_improvement = geomean(performance\_improvements) \times geomean(energy\_improvements)$$

energy_delay_improvement=geomean(performance_improvements)×geomean(energy_improvements)

We will use the energy-delay improvement to calculate your score according to the following curve:

- Energy Delay Improvement = 10 -> Grade = 60
- Energy Delay Improvement = 25 -> Grade = 80
- Energy Delay Improvement = 50 -> Grade = 90
- Energy Delay Improvement = 80 -> Grade = 100

Rules

1. All of your code should be in kernels.c. Any changes to the Makefile will be ignored. Do not rename any files.
2. You may not write any assembly instructions.
3. No external libraries may be used in kernels.c. (you can write your own functions of course). This also means you shouldn't fork around.
4. Please no hacking the server! We are trusting you enough to run *your* code, so please be nice. It is not an achievement to hack someone when they let you run your own code. : p

Downloading and running the Lab

You can use *any* lnxsrv machine to edit your files (i.e. keep using lnxsrv06 with VScode). But for running the code, you should use *lnxsrv07* for this lab, as the cache profiling only works on this machine. Having some/most people use lnxsrv06 for editing code would actually be helpful, because it would help spread out the load between lnxsrv06 and lnxsrv07.

Step-1: Go to the lab website on your browser. Click into the "Request Lab" button on the webpage, insert the requested information to get a tarball. Then Secure copy(upload) the tarball from your local machine to the seas machines(using scp command on the command line or using other file transfer tools such as CyberDuck). In the terminal you can do the following:

```
scp <yourlocalfile> <yourusername>@lnxsrv07.seas.ucla.edu:~/<yourdir>
```

Step-2: Log onto the lnxsrv07 server either using a terminal or your favorite text editor (VScode, etc), then go into the directory where the tarball was uploaded. Then untar the tarball. Note: YOURNUMBER should be replaced with the actual number assigned to your tarball.

```
ssh <yourusername>@lnxsrv07.seas.ucla.edu
tar -xvf target-YOURNUMBER.tar.gz
cd target-YOURNUMBER
```

Step-3: Now you can start optimizing the code contained in kernels.c. Once you're done optimizing, you can check the correctness and the performance of you code by "making" your code and running the following commands:

```
make
./perflab
```

Make sure not to type "make perflab" or "make ./perflab" all one one line, because this will cause you to not compile everything (there's another binary called "plain" that needs to get compiled as well).

You can also run a specific test input with the -i flag, this one runs test 2:

```
./perflab -i 2
```

The lab will generate a report for you that indicates your performance. If you run “locally” like above, you will run on lnxsrv7, which will only give you an approximate grade, since it is not the machine the grading server runs on.

Step-4: In order to be graded, you need to submit your work to the grading server, to do so, simply do:

```
make submit
```

The server will give you a similar version of the performance statistics that you get by running locally. This new submission will also be reflected on the scoreboard. We recommand testing your code for correctness on lnxsrv7, and then submit your code for grading for accurate performance analysis. The grading server only runs one submission at a time on the machine, so there can be some delay before we run your code. The current submission queue approaximate wait time could be seen by two ways: 1.on the scoreboard 2.by typing:

```
make ping
```

in your target directory. We recommend ping the server to see the current workload before submitting, in case the server is loaded , please try to submit later.

Note:

1. If you want a comment to be shown on the scoreboard, you can write your comment in the comment.txt file, please keep it below 25 characters. If you attempt to insert a super long string, that will be rejected by the server.
2. We will count your best submission for your grade. You can submit multiple times, and we will take your best one. The score on the scoreboard is the score you will get (unless we found you didn’t follow the rules) … except that extra credit can change as more people submit (see below).
3. The cache profiling code only works on lnxsrv07, as it is specific to the processor on that machine. Thus, we recommend using lnxsrv07.seas.ucla.edu for correctness and if you want to test the performance of your code before submitting.
   You can use any lnxsrv machine to log into for VSCODE. Load balancing is helpful, thank you!
4. Make sure you save your progress often, things can go wrong at any time, you always want to have a backup of previous editions of your code. Check out version control tools, such as Git.
5. Make sure you are using the latest version of GCC (follow the directions here). You probably did this already for an earlier lab, so no need to do it again if you did.

Interpreting Output

After you optimize your code a bit, you might see the following output:

```
T#  Kernel|   Ni    Nj      Nk  S|Time(ms)  CPE #Inst(M) #VecInst(M)|#L1(M) #L2(M) #L3(M) #Mem(M) MemEn(j)|Speedup
Energyup
 0 Transp.|10000 10000       -  -|  296.4  5.92    418.5         0.0| 273.8   10.3   9.66    1.36  0.03318|    1.8
2.7
 1 Stencil|  128   128     128  8|  306.8  0.57   1792.1       267.5| 616.4   36.5  13.06    0.01  0.04111|   13.9
8.3
 2 Stencil|   64    64      64 20|  589.2  0.56   4016.3       523.0|1301.1   26.4   1.35    0.00  0.06841|   13.6
18.6
 3 Stencil|    4     4 1048576  2|  411.2  6.12    295.9        33.9|  93.6   54.8  26.88    8.70  0.11056|    2.0
1.3
Geomean Speedup: 5.10
Geomean Energyup: 4.79
Energy-Delay Improvement: 24.46
Grade: 79.3
```

The first set of columns show the test number and kernel name. The next set of columns show the problem parameters. Transpose only has two params. Then the next parameters are:

- Time(ms) – Total execution time
- CPE – Cycles per Element
- #Inst(M) – Total instructions executed (dynamic instruction count), in millions
- #VecInst(M) – Total SIMD/vector instructions executed, in millions
- #L2(M)/L3(M)/L4(M)/Mem – Number of L1/L2/L3/DRAM accesses in Millions
- MemEn(j) – “Memory Energy” in “Joules”
- Speedup – Speedup of your program compared to the original code run on our grading server
- Energyup – Energy Improvement of your program compared to the original code run on our grading server

After that is the geomean performance and energy improvements, and grade estimate, as discussed earlier.

Hints and Advice

Things you should do for sure, and like right away:

- Inline all the functions! This reduces function call overhead, and enables the compiler to reason about the program region.
- Create a special version of each kernel that is specific to the matrix dimensions in each test case. This has two benefits: the compiler can create simpler code because it knows what the matrix dimensions are, and you can also optimize for each kernel differently. So what you should do, is in the compute_stencil function, check the input arguments to see if it matches one of the test cases, then call a version specific to that test case.
- Try different loop orderings. Think about which ordering gives you the best spatial locality. Spatial locality is good for cache performance **and** the compiler will naturally vectorize loops with good locality.
- You can also try adding the “restrict” keyword in the first (outer) dimension of your array – this tells the compiler that the corresponding memory arrays don’t overlap, and could make the compilers job easier.
    - Syntax: `func(float a[restrict N][N][N] ...)`

General Advice:

- Every time you try something, check the stats (#instructions, cache accesses, etc.). See if what you did makes sense in the context of those stats – that will help you build up an understanding of what is working and why. I.e. I did loop tiling, and the number of L2 cache accesses went down because there’s better spatial locality for L1. (or maybe it didn’t go down, meaning you didn’t tile correctly, or the tiling size is wrong, etc.)
- You’re going to see a lot of performance variation on multiple different runs. This is unavoidable – we are running on a real machine where OS-level events can affect your performance, as well as contention for resources from other running programs.
- Sometimes, simpler is better for the compiler.
    - Some optimizations can be done effectively by the compiler, so you should only do those yourself in the code if the compiler is not doing a good job. I.e. if you try something and it doesn’t help, then roll back and keep the simpler version! (PLEASE don’t show us your ugly code, that will make us very sad.)
    - E.g. we showed you how to unroll, use separate accumulators etc… but the compiler is like really good at these most of the time. Sometimes, unrolling can make things worse for the compiler because the code gets more complex.
    - Overall, don’t forget to simplify when your optimizations don’t work. As Donald Knuth said, *“Premature optimization is the root of all evil in programming.”*
- Sometimes a recommended optimization (i.e. in the hints section) may slightly lower the performance for a particular case. This doesn’t mean it’s a bad optimization, it just means there was some interaction with your program and the compiler, or maybe there was some performance variability between runs. Keep going and

implement the rest of the suggested optimizations, and you will get a good score. Also, you may need to play with the parameters of the optimization for it to work well (e.g. tile size).
- When lnxsrv07 is highly loaded, running locally will not produce accurate results – this will likely be the case near the deadline of this lab. However, our grading server is not running on lnxsrv07, and it will report accurate results (with some variation of course) – so you can use this to measure your performance if you like. However, there will be a wait-queue, because only one person can run at a time.
- Start early, so you have enough time to finish and submit to the server, as the deadline approaches, there will likely be much longer wait times for submission.

Things you could do *but probably shouldn't do* unless you're a stubbornly curious person and you have time to burn:

- Manually unroll the code and/or use manual accumulators because you think you know better than the compiler. (But maybe sometimes you accidentally get lucky and do help the compiler)
- Try to use tiling to improve temporal locality as well (for stencil I mean). Intuitively, tiling can prevent you from going down a really long dimension without getting any reuse in your caches. What should your tile size be? Try to calculate the size of all the data you access within some inner loops – make sure this is less than the cache size you are targeting.
- The compiler will try to vectorize your code, and it's kind of interesting to see why it did or didn't vectorize. You can dump the (ugly) vectorization report by passing the flag `-fopt-info-vec-all` to GCC when compiling stencil.c. There is a comment in the Makefile that points out where to add the flag. Just look for the line number of your loop to see if it got vectorized or not. Sometimes it might vectorize an outer loop.
- Go look at the generated code by opening up the assembly of kernels.o (it's in the build/ folder) using objdump or gdb. You might find out that certain program transformations help the compiler generate better code. However, looking at the assembly for these programs at high optimization levels may drive you to madness. So really proceed with caution.

CPU Microarchitecture Parameters

The machine we're using is the: Intel(R) Xeon(R) CPU E5-2640 v2

It shouldn't be necessary to know the specific CPU properties to get a good grade; however, it could be useful under some advanced optimizations, and also just for understanding your results in general.

Compute

This macine supports 256-bit wide vector instructions called "AVX". This means that it's possible for it to use a single AVX instruction to do 8 float multiply-accumulates per cycle (as 8 * 32-bits = 256).

Cache

The cache block size is 64 bytes. Here's the size and associativity of the three levels of cache:

- L1: 32KiB & 8-way associativity
- L2: 256KiB & 8-way associativity
- L3: 20MB & 16-way associativity

Extra Credit

The main purpose of the scoreboard is so that people can compare their performance against others to see what opportunities there might be on different kernels.

But, we will also give some extra credit to students who want to be at the top of the leaderboard. The top 20% of submissions will earn some extra credit, but to avoid any steep dropoff in bonus, we'll scale the bonus linearly down

from the maximum. More specifically, the best score will get an extra credit bonus of 20%, the top 1% will get a bonus of about 19%, and so on down to only 1% bonus for the top 19%.

Please note that your extra credit score is not final until all students make their final submissions.