# High Performance Multigrid V-Cycle

Shashank Obla, Eric Tang
*Department of Electrical and Computer Engineering*
*Carnegie Mellon University, Pittsburgh, PA*
{sobla, erictang}@andrew.cmu.edu

***Abstract*—Partial Differential Equation (PDE) solvers form a large fraction of many scientific computation runtimes. An iterative solution to the Poisson equation, a common PDE, involves stencil computations which have poor memory reuse and hence are a challenge to accelerate. The multigrid method further increases complexity by requiring efficient kernels for various grid sizes and kernels to transform between the grid sizes. In this work, we analytically design kernels for the important kernels in the V-Cycle multigrid solver using the Jacobi smoothing function. Our kernels are designed to achieve peak throughput of the bottleneck instructions but are subject to inefficiencies due to memory bandwidth limits and the compiler. However, they still achieve at least 50% of the peak. We also design an appropriate intermediate grid layout and put our kernels together to implement the full V-Cycle flow. Compared to our chosen scalar baseline, which is already 40x faster than a PDE solver library, our implementation provides up to 10x speedup. We also discuss a parallelization scheme to achieve multi-threaded performance while minimizing the parallelization overhead.**

## I. Introduction

Partial differential equations (PDEs) such as the Poisson Equation commonly occur in the natural sciences from electrostatics to fluid dynamics. For example they are used to obtain the electric potential given a charge distribution. Though some of the simple setups can be solved analytically, complex situations as found in simulations require a numerical solution of the equation and are computationally intensive needing acceleration. The Finite Difference Method [1] (FDM) is one of simpler techniques that uses the finite differences approximation of a differential to express the possibly non-linear PDE as a system of linear equations which can be solved using matrix algebra techniques. The techniques fall under the class of Iterative Stencil Loops, useful not only to solve PDEs but also in classical Image Processing. The Poisson equation is of the form:

$$\nabla^2 \phi = \rho$$

In this project, we explore the V-Cycle Multigrid method [2] to solve the 2D Poisson Equation.

### *V-Cycle Multigrid*

The multigrid method uses a hierarchy of resolutions to improve the convergence of problems exhibiting multiple scales of behavior, for which other methods have variable convergence rates. A single iteration of the multigrid approach includes multiple computation kernels most of which are different stencil computations:

- **Smoothing**: A traditional kernel in iterative methods, such as the Jacobi or the Gauss-Seidel relaxation update performs the main computation to obtain. These can be used in on their own to also solve the PDE but are expected to converge slower.
- **Residual Computation**: Also a stencil computation similar to the smoothing, the residual computation calculates the error between the calculated and the required RHS in the equation.
- **Restriction**: This step downsamples the residuals into a coarser grid similar to reducing resolution of an image using another stencil.
- **Prolongation**: Also called interpolation, this step reverses the restriction to reconstruct the correction into a higher-resolution grid.
- **Correction**: Updates the solution with the computed correction as a matrix addition.

In order to obtain high performance Multigrid V-Cycle, we write high performance implementations of these kernels and then compose them to perform the entire computation. We implement the above kernels using AVX and AVX2 SIMD operations on the Intel x86_64 architecture, specifically the Haswell $\mu$arch. Each kernel is benchmarked separately as well as part of the complete V-Cycle Multigrid method.

The Proto library [3] from Berkeley Labs as our functional baseline. The Proto library was designed for writing efficient solutions to differential equations. We will compare to their MultiGrid and pointRelax examples.

## II. Kernels

The important parameters for our kernel are the number of input elements, number of output elements and the number of registers that will be used. We are computing on double precision floating point numbers which means we can use the packed double SIMD instructions for the most part and store a maximum of 64 elements in the 16 SIMD registers shared amongst the input and output elements. We decide the kernel size to be at least large enough to fill all the pipelines of the bottleneck instructions to achieve peak throughput and increase it until we can use all the SIMD registers.

### A. Jacobi Update

The core computation of the smoothing kernel is the application of the 5-point Laplacian stencil. Each kernel will compute 48 output elements which are stored in 12 SIMD registers. These output elements come form a 3x16 block of elements of the output. In order to compute 48 output elements 16 inputs are required from a 1x16 row of the input. Even though an registers is required to hold the values of the 5-point stencil, this is still possible since the output element can overwrite input register once that register has been used to calculate 3 SIMD outputs.

```
void compute_laplacian_fivepoint(double *in,
 ↪  double *out, int N, double c) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            // Periodic Boundary Conditions
            int i_p = (i+N-1) % N;
            int i_n = (i+N+1) % N;
            int j_p = (j+N-1) % N;
            int j_n = (j+N+1) % N;

            out[i*N+j] = c*(in[i_n*N+j] +
 ↪  in[i_p*N+j] + in[i*N+j_n] + in[i*N+j_p] -
 ↪  4*in[i*N+j]);
        }
    }
}

void jacobi_update(double *phi, double *L_phi,
 ↪  double *rho, int N, double c) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            phi[i * N + j] += L_phi[i * N + j]
 ↪  - c * rho[i * N + j];
        }
    }
}
```

In more detail, the steps are as follows

1) Set 12 SIMD registers to all 0s. Load a 3x16 from rho, multiply these elements by -c and add them to the output registers.
2) Load the first 16 elements from the input matrix and the 4th row of 16 elements from the input matrix. Multiply the first group by the appropriate stencil values and them to the first 16 elements of the output. Perform an FMA with the second group of elements and the stencil to calculate the output from the 2nd and 3rd rows of the output.
3) Load the 2nd 16 elements from the input matrix and the 5th row of 16 elements from the input matrix. Perform an FMA with these inputs and the stencil to compute 12 independent output elements once again.
4) Load 3rd group of 16 elements from the input matrix and use it to calculate 12 independent output values.
5) Load the elements offset by 1 to the left to add to the 12 outputs using the (1,0) stencil value. Repeat this with the elements offset by 1 to the right to add to the 12 outputs using the (1,2) stencil value.

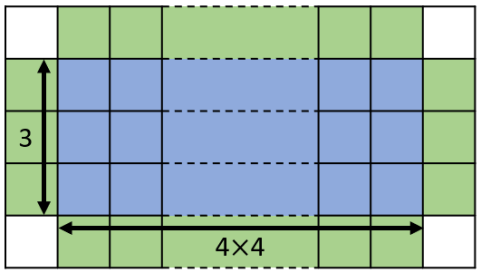


Fig. 1: 3x16 Jacobi Kernel

In this description, the five-point laplacian and jacobi update is fused into one kernel. The increment that is to be done to the original grid can be added as a +1 to the center point of the original five point stencil. This thereby helps us to reduce the number FMA operations by 2. Furthermore, for the given application that we are working with, the laplacian_fivepoint function will always have a value of c=0.25 which allows us to remove the center point of the five-point stencil upon simplifying the computation (1-4*0.25=0). However, due to this realization near the end of the project we were unable to use this insight to redesign the kernel.

### B. Residual Computation

```
void compute_residual(double *phi, double *rho,
 ↪  double *residual, const int m,
    const int n, const double c) {

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            // Periodic Boundary Conditions
            int i_p = (i+N-1) % N;
            int i_n = (i+N+1) % N;
            int j_p = (j+N-1) % N;
            int j_n = (j+N+1) % N;

            residual[i * n + j] = rho[i * n +
 ↪  j] - (phi[i * n + j_n] + phi[i * n + j_p]
                + phi[i_p * n + j] + phi[i_n *
 ↪  n + j] - 4 * phi[i * n + j]) * c;
        }
    }
}
```

We designed multiple version of the kernel to increase memory reuse, as, even though FMAs are the compute bottleneck, memory loads become a bottleneck if not reused enough.

*1) Simple 5x8 Kernel:* This kernel design is very similar to Jacobi kernel in that it computes a 5x8 tile by loading all the required inputs to compute them. It maintains the requirement of 40 independent operations to achieve peak throughput of the FMA instructions. The reuse in this kernel is for the middle section where one loaded element can be used to compute a part of 3 outputs. The size was limited to 5x8 because, a couple of the registers are used to store the constants required for the computation and others are used to store and reuse the

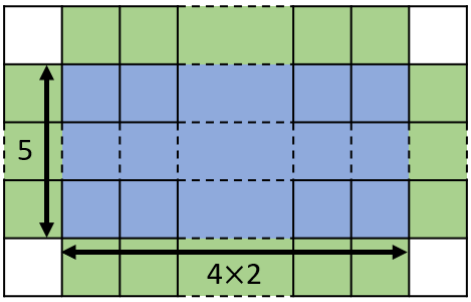


Fig. 2: 5x8 Kernel

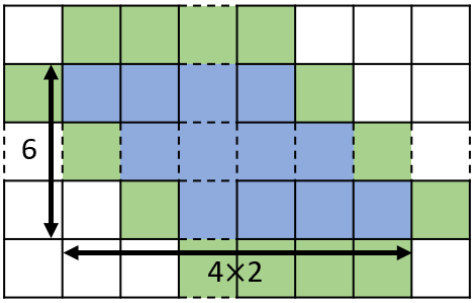


Fig. 3: Skewed 6x8 Kernel

inputs. To feed this kernel we also pad the input matrix to make sure maximum of our loads are aligned (in the current state, 3 out of 5 loads are aligned).

*2) Skewed 6x8 Kernel:* We experimented with skewing the iteration space to make the reuse more apparent at the cost of completely losing the little alignment in memory accesses we had with the previous kernel. This kernel as shown in Figure 3, is able to reuse the inputs required to compute the four outer elements of the 5-point stencil. For the top-most and bottom-most set of elements there is some lack of reuse and also the inputs loaded for the center element of the stencil are not reused. This kernel performs worse than the 5x8 kernel, hence not used in the final implementation.

*3) Sweeping 3x16 Kernel:* This kernel reimagines a simple 3x16 kernel to make sure that the 3x reuse, characteristic of the stencil, can be maintained throughout the computation. At every step during the steady state of the kernel, the kernel loads a new row of elements, loads the center set of inputs, resues it for all the 3x6 elements. Following this it completes the top row of elements, by loading the inputs required for the left point of the stencil, partially computes the middle row for the right point in the stencil and finally stores out the top row. It then repeated this through a column of the grid, with appropriate prologue and epilogue to handle the first and last two rows.

Even though this kernel showed promise, the natural way of writing this kernel causes the compiler to generate a lot of extraneous instructions to handle the register

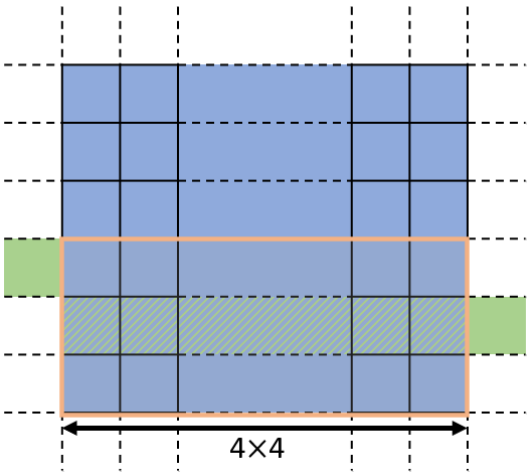


Fig. 4: Sweeping 3x16 Kernel

management, which could be potentially ameliorated by manually unrolling the loops. Given the time constraints, the final version of this kernel was subpar to the simple version and hence wasn't used in the final implementation.

### C. Max-Norm

```
double compute_max(const double *in, const int
 ↪  N) {
    double max = 0;
    for (int i = 0; i < N; i++) {
        max = std::max(max, std::fabs(in[i]));
    }
    return max;
}
```

A part of the algorithm is also calculating the max-norm which is a reduction operation. We can implement the reduction in parallel until we get down to the size of the kernel at which point an epilogue can compute the final max element output. At each step we reduce our input matrix by 1/4, 2 in each dimension.

The kernel loads a $8 \times 8$ block of elements into the 16 SIMD registers. First 8 SIMD `max` operations are performed to reduce to 8 elements followed by further 4 SIMD `max` operations to reduce to 4 final output elements. Since the `max` instructions needs only 3 independent operations we are always filling up the pipeline. We can further unroll the outer loop to start loading the next set of elements to overlap with the next computation. Absolute value is computed by taking a bitwise `and` with `0x7FFFFFFFFFFFFFFF` to set the sign bit in the double to 0. The bottleneck in the kernel is the max instruction which has a throughput of 1, that is 4 max operation on doubles.

### D. Restriction

```
void restrict_residual(double *residual,
 ↪  double *residual_coarse,
    const int m, const int n) {
```

```
    for (int i = 0; i < m / 2; i++) {
        for (int j = 0; j < n / 2; j++) {
            residual_coarse[i * n / 2 + j] =
 ↪  0.25 * (residual[2*i*n + 2*j] +
 ↪  residual[2*i*n + (2*j+1)] +
 ↪  residual[(2*i+1)*n + 2*j] +
 ↪  residual[(2*i+1)*n + (2*j+1)]);
        }
    }
}
```

The restriction kernel computes 8 elements of the output by loading a 2x16 set of input elements. First the input elements are shuffled and permuted appropriately so that all the elements in the first 2x2 block of input are in the lowest 64 bits of 4 different SIMD registers. This is repeated for the other 3 2x2 blocks to create 4 SIMD vectors which, when reduced with FMAs, produces the first 4 elements of the output. This is then repeated similarly to produce the next 4 elements. As is clear from this description, we are simply reducing using a dependent chain of FMAs, but we have independent shuffles and permutes which allows us to get close to peak for the bottleneck instructions.

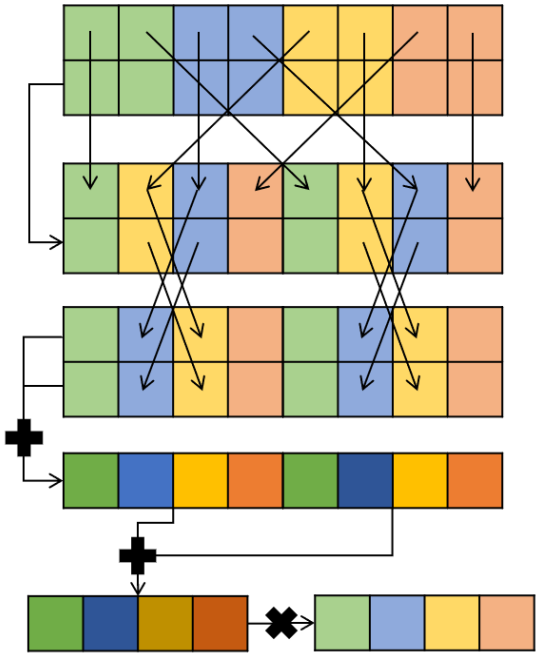

Fig. 5: Restriction Kernel

### E. Prolongation and Correction

```
void correction(double* phi, const double
 ↪  *correction_coarse,
    const int m, const int n) {
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            phi[i * n + j] +=
 ↪  correction_coarse[(i/2) * (n/2) + (j/2)];
        }
    }
}
```

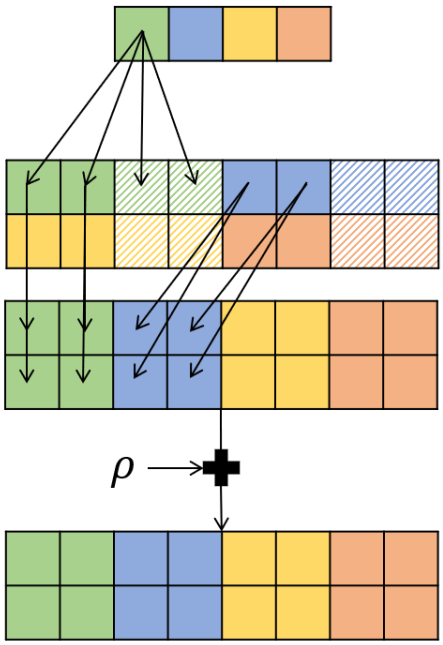


Fig. 6: Prolongation Kernel

This kernel is quite simple in that it broadcasts loads every element from the input matrix and loads elements from the previously calculated Φ matrix, adds them and stores them back to produce the corrected output Φ matrix.

The kernel computes 32 output elements using 8 SIMD registers to store the broadcasted input as two rows of 16 elements each and 8 other SIMD registers to load the Φ matrix, as destination of the add instruction and store the output. The permute instructions or alternatively insert instructions are used to reorder the inputs to the 2x2 format required for the output of this kernel and then added with the $\phi$ matrix to update the correction. The `add` instructions are the bottleneck, but are all independent and we're able to fill the pipeline.

## III. Composing Kernels for High Performance

To get high performance, each kernel uses a padded version of the input in order to resolve the boundary conditions more easily. Since each kernel requires the padded inputs, there is no need to remove the padding ,between kernel inputs and outputs. In addition to this, the padding on the left and right sides of the grid are padded such that there are four extra columns on each side. This padding allows us to perform more aligned SIMD loads within each kernel rather than being forced to use unaligned SIMD loads for all elements.

In addition to the padding, macro instrinsics are used so that the compiler uses the proper instructions within each kernel. Without these instrinsics, the compiler tried to optimize away some unaligned SIMD loads and use vinsertf128 instructions instead. However, since we have

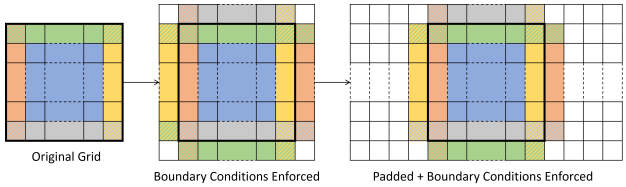


Fig. 7: From left to right, the padding that needs to be added to the original grid can be seen. Starting with the original grid on the left, with a periodic boundary condition, in order to compute the outputs an additional row and column must be added along the edges. Following this, additional padding is added to allow for simple alignment for SIMD loads and stores.

written these kernels such that it is accessing data that has been used very recently, we know the data resides in the L1 cache. For this reason, it is cheaper to simply perform the load operation rather than using this instruction which will try to do a scalar load and then shift the data around within the SIMD registers.

By designing the kernels such that they use the hardware available as efficiently as possible we are confident that we have good performance but would not necessarily say that it is high performant. From investigating the assembly from the kernel, it can be seen that the memory address calculation is not being done as efficiently as possible. The memory addresses to be loaded from are being pushed to the stack. This leads to two memory requests needing to happen for some loads - one to load the address, one to load the data. Since the Jacobi update kernel is the most used and is what the most runtime is spent on, this improvement could lead to a large increase in performance overall.

## IV. Parallelization

The multigrid V-Cycle algorithm can be parallelized by splitting up the initial grid into blocks, each of which can be worked on in parallel. From the work that was done to design the kernels for this algorithm, we can see that each section of the grid can be worked on independently. For example, with 4 threads, the grid can be split up into 4 quadrants. Then each of these 4 quadrants can be treated as a grid and the entire V-Cycle can be done on this fraction of the grid with only the data from the edge (between the tiles) cells communicated between the adjacent threads after every iteration of the Jacobi kernel. We can further reduce this communication by communicating more than one column/row of elements and recomputing the elements to reduce the need for synchronization at the cost of recomputation.

We can also pack the grids into a tiled format to ensure memory contiguity and exploit the fact that each thread can retain it's section of the data in its cache throughpout the recursive calls in one iteration of V-Cycle.

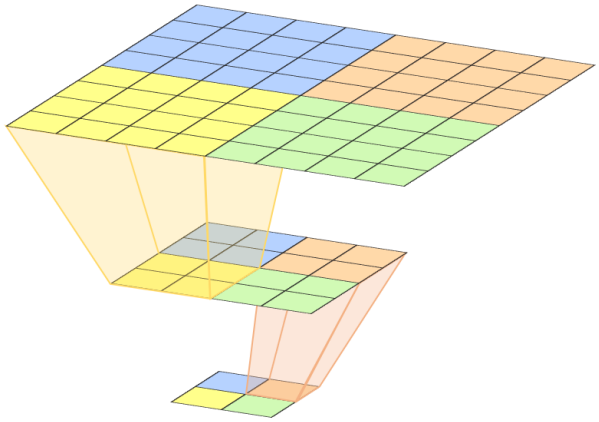

Fig. 8: The grid should be divided up such that each thread works on a different portion of the grid. Each color represents the points that one thread should work on in the V-Cycle algorithm.

## V. Results

To evaluate our implementation, we first evaluate the efficiency of each of the 5 kernels that we have implemented. Following this, we look at the performance of the entire Multigrid V-Cycle as a whole and compare to a scalar C++ baseline implementation that has been checked to match perfectly with the original baseline Proto implementation. This scalar C++ implementation already achieves a speedup of around 40x over Proto; Proto being slower due to the object-oriented implementation overhead, while our implementation lacks any flexibility in implementing the full feature set of Proto. The results for each kernel can be seen in Figure 9.

### A. Jacobi Kernel

For this kernel, peak performance is identified as the maximum throughput of the SIMD FMA instruction which gives 16 FLOPS/cycle. The kernel that we have designed obtains 56% of peak. After analyzing the assembly from our kernel, we noticed that the address calculation that is done to load from the input grid is not being handled as efficiently as possible. Despite expanding all arithmetic that needs to be done to calculate the offset to each row and column, the compiler pushes the load address to the stack. This thereby doubles the number of memory accesses that need to be done for each of these SIMD load instructions. Following this, the compiler also utilizes poor addressing modes for these load instructions that can also lead to degraded performance in this kernel.

### B. Residual Computation Kernel

Peak performance is identified as the maximum throughput of the SIMD FMA instruction which gives 16 FLOPS/cycle. This kernel achieve 58% of peak and

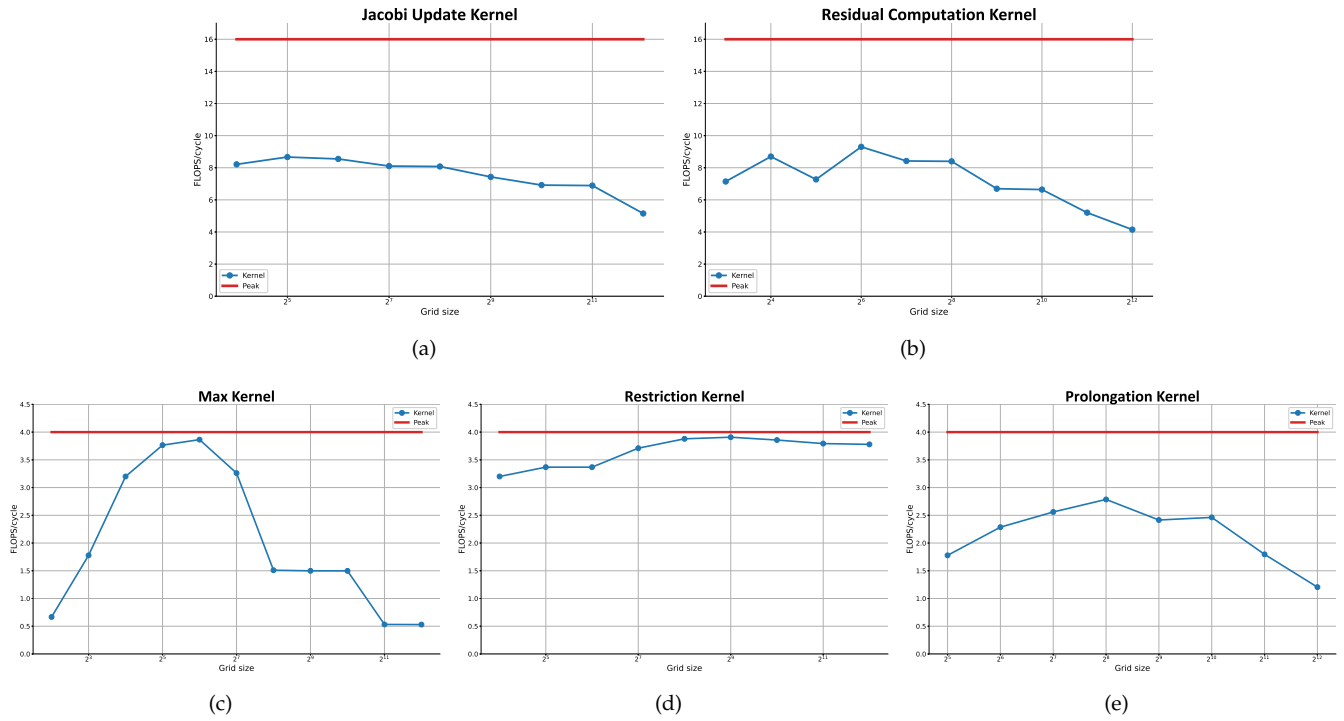


Fig. 9: In these graphs, we show the performance of the various kernels that have been implemented for various grid sizes. The sizes of the grid for each kernel vary from 16 - 2048. In blue, the performance of the SIMD kernel implementation is shown, while the horizontal red line shows the peak for our CPU architecture.

suffers from many of the same shortcomings as the Jacobi kernel. Since both of these kernels require the same five point stencil operation it follows that both of these kernels achieve similar performance.

*Max Kernel:* In this case, the results shown in Figure 9c were generated by repeatedly running the maximum computation kernel over the entire grid. Peak performance is identified as the maximum throughput of the SIMD MAX instruction which gives 4 FLOPS/cycle. This kernel is able to achieve 95% of peak.

### C. Restriction Kernel

In Figure 9d the results from repeatedly running the restriction computation kernel on the same few rows of the input grid are shown. Peak performance is identified as the maximum throughput of the SIMD permute instruction which gives 4 FLOPS/cycle. This kernel is able to achieve peak as is shown in the graph.

### D. Prolongation and Correction Kernel

This kernel's performance which is shown in Figure 9d was generated by repeatedly running the correction kernel on the same few rows of the input grid. Peak performance is identified as the maximum throughput of the SIMD permute and add instruction which gives 4 FLOPS/cycle. This kernel was able to achieve 68% of the peak throughput.

### E. Multigrid V-Cycle

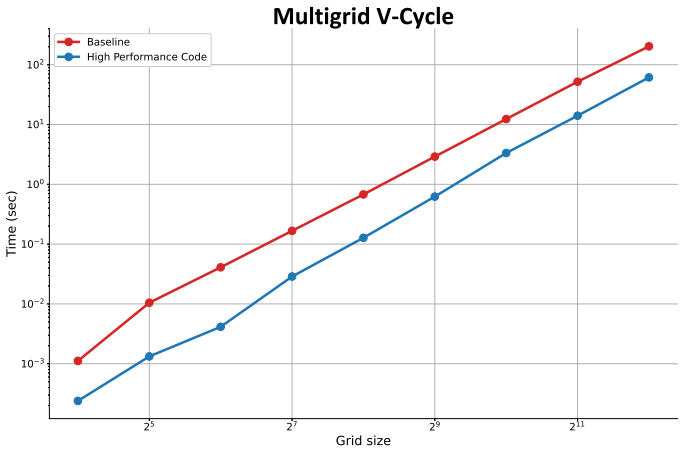


Fig. 10: In red the scalar Multigrid V-Cycle implementation is shown while the blue shows our own fast code implementation. As we scale to even larger sizes, the fast code implementation maintains a speedup over the baseline of around 3x.

The results from Figure 10 was obtained by running the iterative Multigrid V-Cycle algorithm for 100 iterations with a NxN grid. All of these results were gathered on `ece004` from the ECE number clusters. We achieved a

speedup ranging from 3.3 - 9.8x over the scalar baseline code. For smaller sizes, the problem is able to fit in the higher-level caches and thus we are able to achieve larger speedups over the baseline. However, even for larger sizes, the speedup is consistently around 3.5x faster than the scalar baseline.

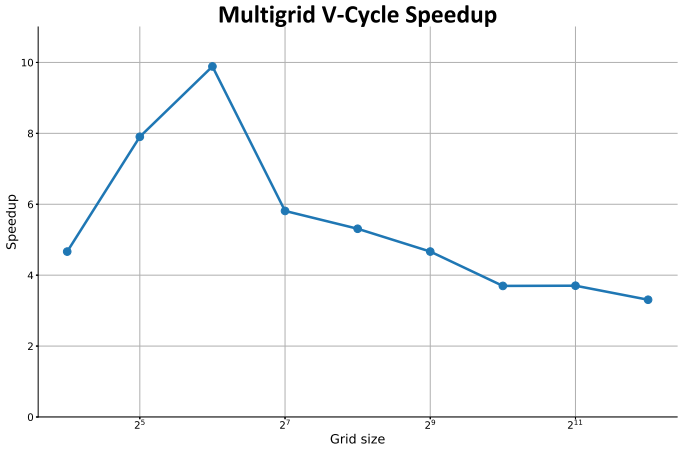


Fig. 11: Speedup of our multigrid implementation over a scalar baseline

## VI. Future Work

One optimization that can be done is removing the dynamic memory allocation that happens within the Multigrid V-Cycle recursive calls. Instead, all of these memory regions could be allocated beforehand and each call to V-Cycle then uses the appropriate address depending upon the grid size.

As discussed earlier, another avenue for improvement is rewriting the Jacobi kernel entirely in assembly. From investigating the assembly from the kernel, it can be seen that the memory address calculation is not being done as efficiently as possible. The memory addresses to be loaded from are being pushed to the stack. This leads to two memory requests needing to happen for some loads - one to load the address, one to load the data. Since the Jacobi update kernel is the most used and is what the most runtime is spent on, this improvement could lead to a large increase in performance overall.

## VII. Acknowledgements

Thank you Tze Meng for teaching us how to write fast code. Thank you Het for showing and teaching us about this application (and featuring in our video). Thank you to the great TAs Nicholai (and Upasana). This project took 1000 hours.

## References

[1] Wikipedia, "Finite difference method — Wikipedia, the free encyclopedia," http://en.wikipedia.org/w/index.php?title=Finite%20difference%20method&oldid=1126400243, 2022, [Online; accessed 13-December-2022].

[2] ——, "Multigrid method — Wikipedia, the free encyclopedia," http://en.wikipedia.org/w/index.php?title=Multigrid%20method&oldid=1106856721, 2022, [Online; accessed 14-December-2022].

[3] A. N. A. G. at Lawrence Berkeley National Laboratory, "Proto," https://github.com/applied-numerical-algorithms-group-lbnl/proto/tree/proto_spiral, 2022.