Approximating a Multi-Grid Solver

Valentin Le Fèvre*, Leonardo Bautista-Gomez†, Osman Unsal† and Marc Casas†
*École Normale Supérieure de Lyon, France
Email: valentin.le-fevre@ens-lyon.fr
†Barcelona Supercomputing Center (BSC), Spain
Email: {leonardo.bautista,osman.unsal,marc.casas}@bsc.es

***Abstract*—Multi-grid methods are numerical algorithms used in parallel and distributed processing. The main idea of multi-grid solvers is to speedup the convergence of an iterative method by reducing the problem to a coarser grid a number of times. Multi-grid methods are widely exploited in many application domains, thus it is important to improve their performance and energy efficiency. This paper aims to reach this objective based on the following observation: Given that the intermediary steps do not require full accuracy, it is possible to save time and energy by reducing precision during some steps while keeping the final result within the targeted accuracy.**

To achieve this goal, we first introduce a cycle shape different from the classic V-cycle used in multi-grid solvers. Then, we propose to dynamically change the floating-point precision used during runtime according to the accuracy needed for each intermediary step. Our evaluation considering a state-of-the-art multi-grid solver implementation demonstrates that it is possible to trade temporary precision for time to completion without hurting the quality of the final result. In particular, we are able to reach the same accuracy results as with full double-precision while gaining between 15% and 30% execution time improvement.

***Index Terms*—Multi-grid, algorithms, parallel processing, iterative method, approximate computing**

I. Introduction

Multi-Grid (MG) solvers are a class of linear methods [1] that emerged in the 80's to increase the convergence rate of more classical iterative methods. They became even more important with the advent of very complex scientific applications [2], which required very powerful linear solvers. The usage of MG solvers has become a common practice in today's parallel systems due to the good scalability properties of these methods, which have been analyzed and modeled in detail [3]. Also, MG solvers have been reported to display more robustness against silent data corruptions than traditional iterative methods deployed over a single grid [4], which also implies they behave well under reduced accuracy scenarios.

MG solvers rely on a grid of evaluation points that discretize the domain of a continuous differential equation. Typically, MG schemes are defined by coarsening a fine-grain grid until reaching a small set of evaluation points where a direct method can be applied. Solutions obtained on the inaccurate and coarse grids are used on the more accurate and fine-grain levels to accelerate the process of obtaining the final solution of the system. Multi-grid solvers typically consist of three components: The *Relaxation*, *Restriction* and *Interpolation* phases. The relaxation phase applies a few steps of an iterative solver like Jacobi or Gauss-Seidel at a certain coarseness level. The restriction phase propagates the algorithmic state to a coarser grid by means of linear transformations while the interpolation phase maps the coarser estimate to the finer version and adjusts the current solution with the new error.

One common way of orchestrating a Multi-Grid Solver execution is via a V-cycle, where we first iterate on the finest grid, then the second finest grid and so on until reaching the coarsest grid where a direct solve is used instead of an iterative method as the problem size has become smaller. Then we iterate again on all the other grids in the reverse order to have a solution expressed with the initial granularity of the grid. Different parameters, such as the iterative method used at each step or the granularity of the grid and the previously mentioned cycle shape, affect the convergence rate of the algorithm.

In this context, this paper investigates different trade-offs between accuracy and execution time (or energy consumption) for MG algorithms. We focus our efforts on one of the most popular parallel implementations of a multi-grid solver, the BoomerAMG [5], implemented using the HYPRE library [6]. We change the shape of the V-cycle and adapt the number of relaxations at each cycle to achieve performance gains. In addition, we propose a new multi-grid algorithm that dynamically optimizes its floating-point precision to improve performance and reduce energy consumption while maintaining the quality of the final solution. In particular, this paper makes the following contributions beyond the state-of-the-art:

- We evaluate the performance impact of solver's parameters such as the shape of cycles and the number of iterations.
- We propose a new cycle configuration and show that it is more efficient than the state-of-the-art configurations by up to 28.3%. Our experiments consider a total of 9 different matrices and 3 parallel scenarios and demonstrate that our new cycle configuration can be applied to a wide range of problems.
- We evaluate the impact of different floating-point precisions on the time-to-solution while reaching full accuracy on the final result.
- We propose an algorithm that dynamically adapts the precision of the MG algorithm variables to increase performance and efficiency.
- We perform a large evaluation and demonstrate the we can reduce by 15% the execution time needed to reach the same quality result as the original double-precision

algorithm and up to 30% for single-precision accuracy.

The reminder of this paper is organized as follows: Section II introduces the motivations for this work. Section III investigates the impact of different parameters on the time to completion. Section IV explores precision-performance trade-offs. Section V discuss other related works and Section VI concludes this paper.

II. Motivation and Background

In high performance computing (HPC), maximizing performance is the main objective. However, in order to reach extreme scale computing, it is necessary to also improve the energy efficiency of current applications and algorithms. This section explains why approximate computing is a potential solution and why MG solvers are a good candidate for this type of optimizations.

A. Approximate Computing

Approximate computing categorizes a wide set of techniques aimed at trading off computation quality with performance and/or energy. It is based on the intuition that operating at peak level accuracy may produce significant resources waste without adding any valuable information to the numerical data. The increasing restrictions that parallel architectures are suffering in terms of power and circuitry area have made the approximate computing technique an appealing and cost-effective approach that can be potentially applied to a wide range of application domains like data analytics, scientific computing, multimedia or signal processing [7]. Indeed, recent approaches demonstrate how up to 50x energy savings can be achieved when applying approximate computations to the k-means clustering algorithm while only losing 5% classification accuracy [8]. Also, an approximate approach based on neural networks can speedup an inverse kinematics application by more than 20x by allowing a final application error of just 5% [9].

Despite its potential, approximate computing must be applied in an extremely careful way. For example, reducing the accuracy of control flow instructions or memory addresses computations may cause segmentation faults. Also, applying approximate techniques to floating point computations can potentially lead to wrong results or cause iterative algorithms to stall if they are blindly applied. The success of approximate approaches requires them to be judicisouly applied to the most low-accuracy tolerant phases of numerical algorithms. In this paper we apply approximate computing approaches to the MG solver in several different ways. Some of them reduce the amount of computations of certain routines while keeping the accuracy of each computation, like reducing on the number of relaxation iterations (Section III). Other approaches keep the number of computations but reduce their accuracy. For instance, using floating point representations with less bits devoted to store the mantissa (Section IV), only at some points in the computation so that the impact on the final solution stays minimal.

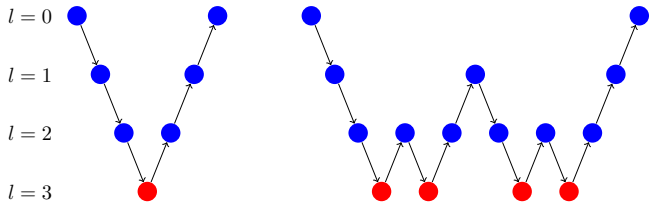


Fig. 1: V-cycle and W-cycle on 4-level grid.

B. Multi-Grid Algorithm

MG's cycles start and end on the finest grained grid and are defined by the order in which its different coarseness levels are applied. The simplest cycle is the V-cycle, already described in Section I, where a few iterations of an iterative method (the *Relaxation* step) are performed at each level from the finest to the coarsest levels and then in the reverse order. Another common approach is the W-cycle scheme where the coarsest grid level is reached several times before going back to the finest grid level. It is possible to generalize the notion of cycle to a k-cycle (where a V-cycle is a 1-cycle and a W-cycle is a 2-cycle). Figure 1 shows a representation of the V- and the W-cyles with 4 levels where we can see how the W-scheme goes back and forth the coarsest level 4 times before reaching again the finest level of the grid. The cycles are executed multiple times iteratively until the error is lower or equal to a certain *tolerance* (i.e., the expected accuracy) or until the maximum number of cycles is reached.

III. MG Algorithm Cycle Optimization

A. Relaxation VS Cycle Shape Study

In this Section we compare different types of cycles and study how the number of *Relaxation* steps influence MG's convergence. The MG algorithm makes use of many different input parameters that have an impact on its success. In our context, we define a strategy as a combination of values of the most relevant parameters: The type of cycle (V or W), the number of relaxation steps α, and the total number of coarseness levels of the grid.

Type of cycle	V	V	V	V	W	W	W	W
α	1	2	3	10	1	2	3	10

TABLE I: Combining α and V/W cycles.

We consider a total of 8 different strategies shown in Table I where we test V- and W-cycles and we perform 1, 2, 3 and 10 relaxation steps for each kind of cycle. For all configurations, the number of different coarseness levels of the grid is 8. To compare the different strategies we consider an input matrix of size 512000×512000, and a maximum number of cycles from 1 to 100. The algorithm's tolerance is set to 0 to always reach the maximum number of cycles (1 to 100 depending on the experiments). We measure for each experiment the final relative residual norm and the execution time. Each experiment is run 10 times to have an accurate average execution time.

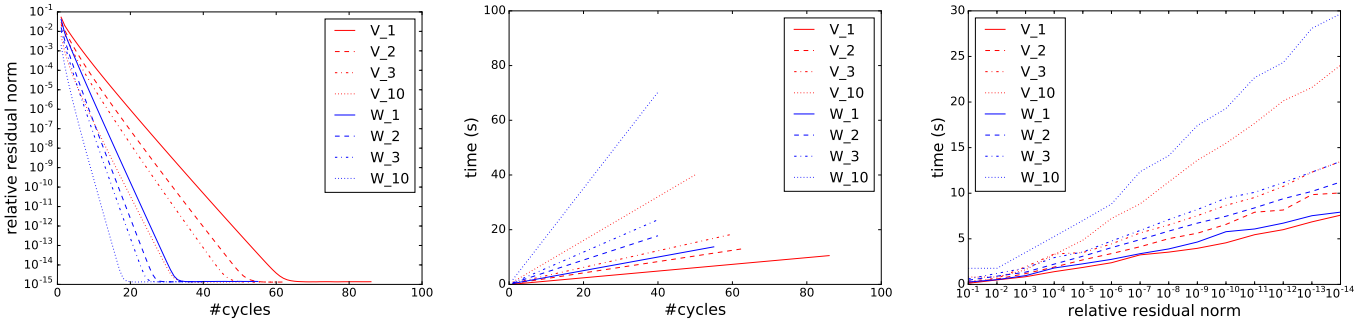


Fig. 2: Final residual norm of the 8 strategies per iteration (left), interpolated execution time per iteration (middle) and convergence time as a function of the tolerance (right).

The results are presented on Figure 2. The left figure shows how the accuracy evolves with the number of cycles performed. For example, we see that the V cycle with 1 relaxation step (plain line in red) converges with more cycles than the other strategies. However, if we look at the middle figure we observe that this configuration converges faster than the other methods. This makes sense as the first configuration is a simple V-cycle with only one relaxation, thus even when the execution requires more cycles, each cycle is less time consuming and therefore the total execution time is shorter.

To be able compare the convergence speed, we present the cost of reaching a given accuracy on the right figure (the right of the x-axis is a small tolerance i.e. correspond to accurate results while the left of the x-axis represent inaccurate (but fast) results). What we can observe is that, as expected, increasing the number of relaxation steps (i.e., complexifying the cycle) decreases the number of cycles needed for convergence but it increases the overall time to do one cycle. We see on the right figure that actually, for a given precision, the simple V-cycle with only 1 relaxation at each step is the fastest way to reach it, followed closely by the W-cycle with $\alpha = 1$.

The conclusion is that relaxation steps seem to be too costly for the accuracy they grant. It is better to do more cycles, thus more moves in the grid, than doing more relaxation steps. This, once again, demonstrates that MG algorithms are faster than classic iterative methods.

B. Cycle Complexity Breakdown

Given the previous results, we embarked to investigate how to improve the efficiency of the simple V-cycle with 1 relaxation step. The first step is to breakdown the time spent in the different parts of a cycle. Note that all the matrices for each level are computed in a setup phase and it is not necessary to analyze that setup time. We only focus on measuring the following two computations: (i) the time spent doing a relaxation at each level and (ii) the time spent computing the next linear system. The latter (i.e., computing the next linear system) can be divided in two options: *going down* by restricting the solution to a coarser grid, which corresponds to a sparse matrix-vector computation; and *going up* by interpolating the error term which also corresponds to another sparse matrix-vector computation.

Level	Matrix size	Non-zero elements	Relax (down)	Relax (up)	Restriction	Interpolation
1	512,000	4,042,520	20 ms	20 ms	15 ms	-
2	256,000	6,475,239	20 ms	25 ms	12 ms	4 ms
3	58,893	2,000,513	8 ms	8 ms	3 ms	2 ms
4	14,285	788,509	2 ms	2 ms	1 ms	0.7 ms
5	4,238	386,333	1 ms	1 ms	0.5 ms	0.2 ms
6	609	53,493	< 0.1 ms	< 0.1 ms	< 0.1 ms	< 0.1 ms
7	69	2,873	< 0.1 ms	< 0.1 ms	< 0.1 ms	< 0.1 ms
8	2	4	< 0.1 ms	-	-	< 0.1 ms

TABLE II: Time breakdown of a V-cycle with $\alpha = 1$.

To study the internal time breakdown of a V-cycle we chose as problem an unstructured domain with some anisotropy (denoted as Unstructured-Anisotropy) of size 512,000 with a 8-level grid. The results of these evaluations are depicted in Table II, along with information on the matrix used at the corresponding level, such as matrix size and the number of non-zero elements. Our first observation is that there is a direct correlation between the time spent on relaxations at each given level and the number of non-zero entries in the input matrix. Most importantly, in these results we observe that relaxations represent $\approx 66\%$ of the total cost of a V-cycle, while the matrix-vector multiplications are only $\approx 30\%$. In addition, we notice that the two first levels are the most expensive ones. It is important to highlight that in the experiments depicted in Figure 2, although different number of relaxations were evaluated, all levels executed the *same* number of relaxations.

C. Level-Dependent Relaxation Tuning

Based on this information, we propose to reduce the execution time of a V-cycle by tuning the number of relaxations differently for each level. More precisely we propose the following two ideas: (i) to add more relaxations in the last levels because their cost is negligible and they could potentially reduce the time to convergence or (ii) to remove some relaxations in the first levels to reduce the computational cost, and see how that affects convergence. We translate these ideas into the four following strategies (based on a 8-level grid):

- *Fast* : no relaxations at level 2.
- *Fast2* : 10 relaxations at level 6.

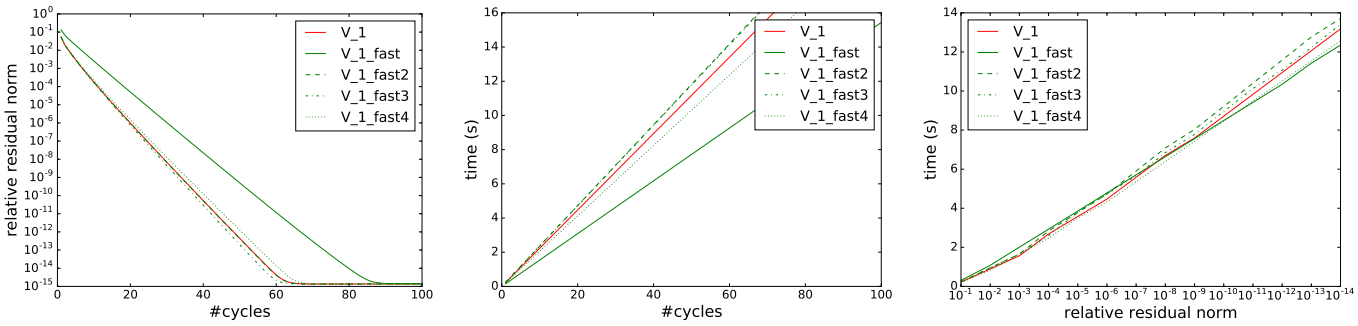


Fig. 3: Evaluation of 4 level-dependent relaxation tuning strategies. Residual norm per cycle (left), time spent in each cycle (middle) and convergence time in function of the error tolerance (right).

- *Fast3* : 2 relaxations at levels 6 and 4.
- *Fast4* : no relaxations at level 3.

The strategy *Fast* aims at reducing the cost of the cycle by removing the penultimate relaxation which is very expensive, and expecting that the accuracy lost at this point can be compensated by the relaxation at level 1. The strategy *Fast2* executes a lot of relaxations at level 6, because it should not increase by much the execution time of the V-cycle. The reason to choose level 6 instead of level 7 or 8 is that the relaxation at level 8 is actually a direct solve. Thus, the result term is almost exact at level 7, because the only source of error comes from the interpolation of e^8 (which is exact) into e^7. This is why, we might expect better results by adding relaxations at level 6. The strategy *Fast3*, pushes the previous idea one step further. If we assume that doing more than one relaxation gets a more accurate error estimation at level l, then at level $l-1$ we do not need to correct a lot by doing more relaxations. However at level $l-2$ we have been through 2 interpolations since the last good estimation of the error vector, therefore we increase the number of relaxations again. Since the first levels are very expensive, we stop this recursion at level 3. Finally, we propose one last strategy *Fast4* which is a softer version of *Fast* where the relaxation at level 3 is removed, producing a less accurate result at that point, but expecting it can be compensated by the two relaxations at level 1 and 2.

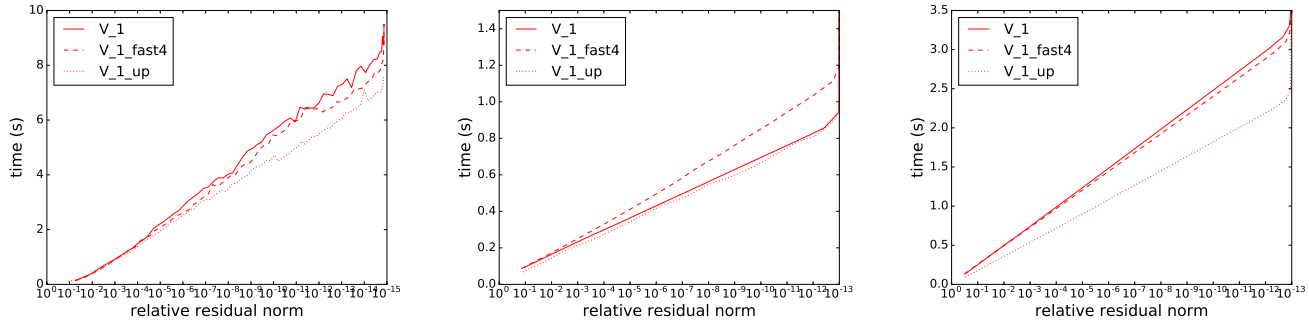
We evaluate these 4 proposed strategies in the original $512,000 \times 512,000$ matrix. The results are shown in Figure 3. The first thing to observe is that removing the relaxation at level 2 (i.e. the *Fast* approach) does not provide any benefit. It does save time during each cycle but the accuracy loss per cycle is too high (i.e., more cycles needed for convergence), leading to a convergence rate close to the baseline configuration. The other thing to notice is that adding more relaxations in the last levels slightly increases the execution time but it does not provide any benefit on the accuracy side. Thus, strategies *Fast2* and *Fast3* are not really efficient. Overall, strategy *Fast* and strategy *Fast4* seem to be more or less equivalent to the original V cycle as it reduces a bit the cost of each cycle and the convergence rate per cycle is also slightly smaller. More tests were performed on a smaller matrix with an initial matrix size of 64,000 with only a 6-level grid. The results were similar (all strategies except *Fast4* were less efficient than the original algorithm) and are not shown in this paper for brevity.

D. An Asymmetric Strategy

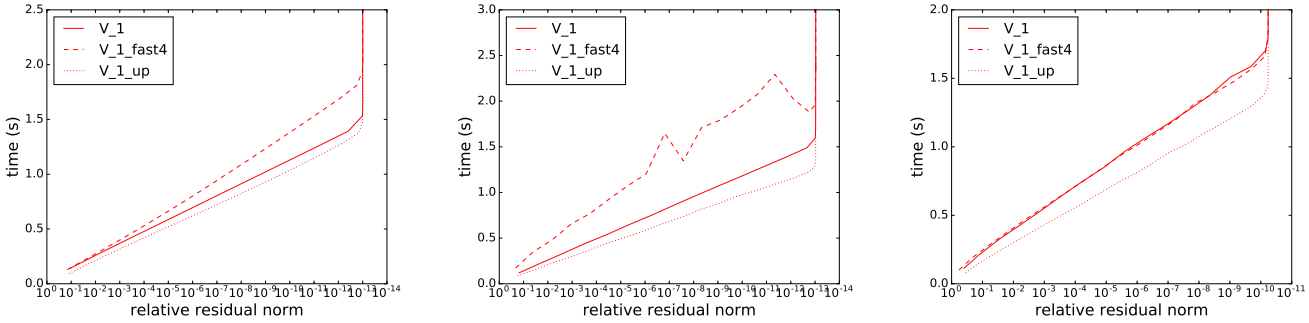
In the previous section, we observed no big improvement compared to the original V-cycle with 1 relaxation at each level, except for strategy *Fast4* which did show some slight improvement. Initially, we applied the same number of relaxations at all levels; then we tried different numbers of relaxations for different levels. However, all these strategies share something in common: for a given level they do the same number of relaxations when going down or up in the cycle, following a very symmetric behaviour.

In contrast, the main idea behind a MG algorithm is rather asymmetric. More precisely, in MG algorithms the types of computations performed when moving to a coarser grid level are different than the type of computations done when going back to a finer grid level. In the first case, the objective is to compute a first approximate solution to the current system while in the second case the objective is to refine the error term. In other terms, we first compute an approximation at level l, then we use the level $l+1$ to compute an approximate error term e^l and finally we redo some relaxation to refine the solution. The two relaxations do not have the same goal.

This analysis opens the door for strategies in which grid levels have a different number of relaxations when going down than when going up in the cycle. In fact, assuming that the values of the error vectors are smaller than a given ϵ from the exact value after just a relaxation step, one would not need to do a relaxation before computing the approximate error term for the next level, but just compute directly the error term when going back up in the cycle. In other words, the relaxations done when going up could potentially compensate inaccuracies obtained after removing relaxations when going down. From that idea we define a new asymmetric strategy: we use a V-cycle in which we do one relaxation at each level only when we are going up in the cycle (i.e., no relaxations when going down). We call this strategy *Up*.



(a) UNSTRUCTURED-ANISOTROPY (b) 3DLAPLACE-7PT(1,1,1) (c) 3DLAPLACE-7PT(0.1,0.1,0.01)



(d) 3DLAPLACE-7PT(5,5,5) (e) 3DLAPLACE-27PT (f) PDE-DIRICHLET

Fig. 4: Comparison of the original algorithm V1 with *Fast4* and *Up* strategies.

We run *Up* and *Fast4*, as long as the classical V-cycle, on the same inputs. For generality purposes, this time we evaluate matrices generated from other physical problems besides the Unstructured-Anisotropy one: i) 3D Laplace equation with a 9-pt stencil (considering three diffrent configurations of the c_x, c_y, c_z anisotropy parameters), ii) 3D Laplace with a 27-point stencil and iii) 3D Laplace partial derivative equation with Dirichlet boundary conditions. All problems use the same matrix size of 512,000. The results are presented in Figure 4.

The first thing we observe is that the *Fast4* strategy does not always perform well. In fact, for two of these four applications the convergence rate is significantly slower than the classic V1 strategy. However, the *Up* strategy seems to be quite efficient; it improves convergence speed by 12%, 7%, 26%, 7%, 20% and 22% for UNSTRUCTURED-ANISOTROPY, 3DLAPLACE-7PT(1,1,1), 3DLAPLACE-7PT(0.1,0.1,0.01), 3DLAPLACE-7PT(5,5,5), 3DLAPLACE-27PT and PDE-DIRICHLET respectively.

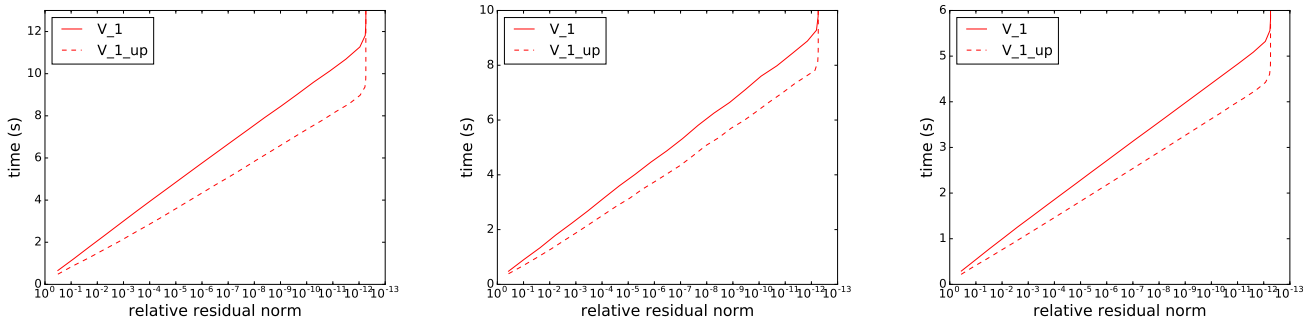
Given these positive results, we extended further our evaluation from single-node runs to distributed memory executions in multiple nodes, in order to test the viability of the *Up* strategy when the algorithm is parallelized and distributed. Larger cases were tested on a cluster with 100 compute nodes, each node equippred with 2 Intel Xeon E5-2630 v3 Haswell 8-core processors, each core at 2.4 GHz, and with 20 MB L3 cache.

The problems tested are 3DLAPLACE-7PT and 3DLAPLACE-27PT. The total size of the matrix is set to either 5,832,000 or 13,824,000, while the topology is composed of either 27 (3x3x3), 36 (6x6x1) or 64 (4x4x4) processors, where each processor holds 1 MPI process and runs 1 OpenMP thread per process. For these 6 possible combinations, we observe an average improvement of 18.4% (ranging from 16.0% to 28.3%) for 3DLAPLACE-7PT and 20.5% (ranging from 16.2% to 25.0%) for 3DLAPLACE-27PT. It seems that *Up* outperforms the classical V-cycle even more when the problem size increases, but seems to cap at around 25% improvement. Figure 5 presents the results for the matrix size 13,824,000 and 3DLAPLACE-27PT, for the 3 different processor topologies. Similar results are obtained for the other physical problems but are not shown here for brevity.

IV. PRECISION-PERFORMANCE OPTIMIZATIONS

A. A new version of multi-grid algorithm

Observing the initial results presented on Figure 2, one can notice that independently of how many relaxations are performed, or which cycle shape is used, and more importantly, of how many cycles are executed, the residual norm of the algorithm reaches a lower bound (it is around 10^{-15}). This bound comes from the internal limitations of the `double` floating-point representation. Indeed, a double uses 8 bytes (52 bits for the mantissa, 11 for the exponent and 1 bit for the sign). Naturally, a space-constrained representation does not



(a) 3x3x3 (b) 6x6x1 (c) 4x4x4

Fig. 5: Comparison of original algorithm V1 with *Up* strategy for 3DLAPLACE-27PT on a 240x240x240 grid with different processor topologies.

allow a full description of all rational numbers (for example between 2^{52} and 2^{53} only integers can be represented). If we increase the number of bits used in the representation we can describe more and more numbers. Similarly, if we reduce the number of bits used in the representation, we lose accuracy for numbers that are not fully representable.

Low precision floating point operations can potentially provide faster and more energy-efficient computations. The reason for these gains is that floating-point units (FPUs) require more circuit logic (silicon for ASICs, LABs/DSPs for FPGAs) for high precision operators (the required area for a FPU increases at least linearly with the number of bits [10]). Thus, reducing accuracy allows a higher number of low precision FPUs, which increases the Instruction-Level Parallelism (ILP) or Vectorization that the processor can reach, hence increasing its peak performance.

Considering our algorithm, lower precision computations could be used if the required accuracy allows it. For instance, if accuracy requirements for the final result were around 10^{-3}, then a 64 bits `double` floating-point representation would not not be needed to reach it and, as such, a lower precision would be enough. Moreover, by analysing the presented results, it is clear that during the first cycles the accuracy is low and it does not require the full precision offered by the double floating-point representation. This observation opens the door for making use of lower precisions temporarily during the first cycles. Therefore, it is important to study the impact of precision on performance for MG algorithms.

To study the trade-off between precision and performance, we modify the relaxations step of the algorithm (note that it is the more time consuming part of the algorithm). In this function we can find 13 internal variables that are originally of type `double`: 5 arrays and 8 scalars. We then propose the following two modified versions of the MG algorithm.

- **AMGfloat**, which changes the type of the 13 variables from `double*` or `double` to `float*` or `float`.
- **AMGmpfr(b)**, which makes use of the library MPFR [11], [12] that introduces a type `mpfr_t`. This type has a parameter b which is the number of bits used in the mantissa of the variable. Every computation is done first in full precision and then rounded to a number with a mantissa using the number of bits given as parameter. In this version of the algorithm *AMG* the 8 scalar variables of the relaxation function are replaced by `mpfr_t` variables, all using the same number of b bits for the mantissa.

Finally, we will denote by *AMG* this original algorithm. In terms of arithmetic precision, *AMG* behaves similarly to *AMGmpfr(53)* and *AMGfloat* to *AMGmpfr(24)*. There can be small differences depending on the rounding method used.

Figure 6 shows the accuracy that can be reached with *AMGmpfr(8)*, *AMGmpfr(16)*, *AMGfloat*, *AMGmpfr(32)* or *AMGmpfr(64)*. The problem used is UNSTRUCTURED-ANISOTROPY, with 2x2x2 processor topology and 20x20x20 matrix size. What we actually see on this figure is the lower bound reached depending on the number of bits used. However, before reaching this lower bound, all precisions show the same accuracy. This means that, for example, while the accuracy of the solution is below the threshold using single precision (32 bits), using 32 or a greater precision to do the same computations will result in the same accuracy. However, we expect single precision computations to be more efficient than higher precisions in terms of time, space and energy.

The second thing we observe is that *AMGfloat* behaves exactly as what we would expect from *AMGmpfr(24)* in terms of accuracy: it shows the same accuracy as versions with more precision in the beginning, then it reaches a threshold between that of *AMGmpfr(16)* and that of *AMGmpfr(32)*. It is important to notice that in the *AMGfloat* version more variables were changed from `double` to `float`, however in the *AMGmpfr(24)* version we do not observe more accuracy degradation. This is because only a few variables from the 8 scalars control the final precision as they are temporary variables used for intermediate computations before being plugged back into the input matrix/vector.

Given this analysis, we design a new algorithm that adapts the precision of the variables during the execution. It is close to the *AMGmpfr(b)* technique except this time the precision

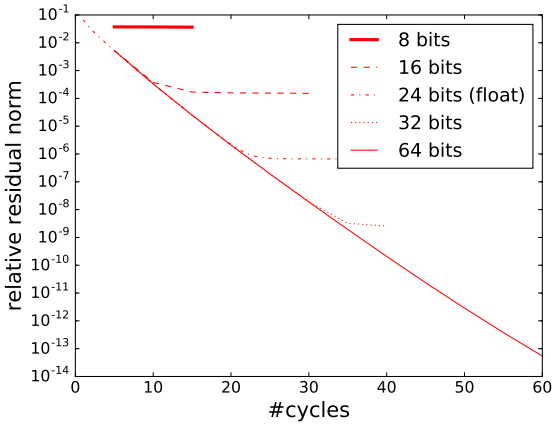


Fig. 6: Accuracy for different number of mantissa bits.

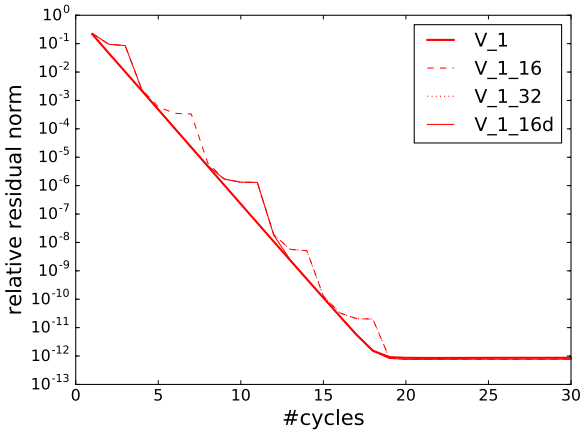


Fig. 7: Accuracy of adaptive algorithms compared to the original double-precision with a precision threshold of 0.8.

can change between two cycles. We fix a threshold on the ratio between the relative residual norm of two consecutive steps to trigger the precision change if the gradient is lower than the threshold (i.e., limited gain in accuracy between two consecutive cycles). Then, we define the following 3 strategies.

- Start at $b = 16$ and do $b = b + 8$ on threshold (V_1_16 on Figure 7).
- Start at $b = 32$ and do $b = b + 8$ on threshold (V_1_32 on Figure 7).
- Start at $b = 16$ and do $b = b \times 2$ on threshold (V_1_16d on Figure 7).

We run these strategies on a 240x240x240 matrix size with a 3x3x3 topology for 3DLAPLACE-27PT. Figure 7 presents the evolution of accuracy for the original algorithm and the 3 new adaptive strategies introduced. We see some steps appear, corresponding to the lower bound on the accuracy at the current precision. Then, the precision is adapted to be able to improve the overall accuracy. Even if we lose some accuracy when waiting for the ratio between two consecutive relative residual norms to reach the threshold; when the precision changes the convergence rate is more important (for one cycle) than that of the original double-precision algorithm (i.e. the slope is bigger), allowing the adaptive strategies to quickly *catch-up* any accuracy loss and reach the maximum accuracy (of 4.7×10^{-13}) in the same number of cycles as the original full precision algorithm (20-21 cycles).

These results demonstrate that adaptive precision can be used to reach the same accuracy in the same number of cycles, while each cycle is expected to be less energy and time consuming because of lower precision. At this point two questions arise. How to evaluate the energy/time savings of this adaptive algorithm in a hypothetical hardware with multiple precisions available? Which precisions should be used to maximize these savings? The complexity to answer these questions comes from the fact that the MPFR library used to do these accuracy experiments introduces a huge overhead in the computation. For instance, running *AMGmpfr(53)* is about 20 times more time consuming than running the classic *AMG*, whereas it represents the same double-precision algorithm. Moreover, this overhead is not influenced by the choice of the number of bits. Therefore, we cannot use the execution times of the computations done with MPFR as representatives of performance differences at different precisions.

B. Optimal set of precisions

In this subsection, we provide an optimal set of precisions to minimize the total execution time, under the assumption that the time evolves linearly with b the number of precision bits: $T(b) = \alpha b + c$. This assumption is supported by current technologies, in which double precision executions take twice the time to complete than single precision executions. This is due to processors having more low precision FPUs than high precision FPUs (See previous section).

Theorem 1. *Given b_{max}, a maximum number of bits, n the number of different precisions $b_1 \leq b_2 \leq \cdots \leq b_{n-1} \leq b_n$ to use, and $T(b) = \alpha b + c$, with α and c two constants, the time to execute a cycle at precision b, then the execution time of our adaptive algorithm is minimized for $\forall 1 \leq i \leq n, b_i = \frac{i}{n} b_{max}$.*

Proof. From the previous experiments we can see that (i) the number of cycles needed to reach the lower bound for a given precision does not depend on the precision used during previous cycles (See Figure 7), i.e. if one starts with cycles at precision 16 bits and then switch to 32 bits, you will need the same number of cycles to reach the lower bound with 32 bits as if you used only cycles with 32 bits from the beginning of the algorithm; and that (ii) the number of cycles needed to reach the lower bound is proportional to the number of bits b used (See Figure 6). We then define MAXITER(b) the number of cycles needed so that the ratio between the relative residual norms computed before and after these cycles is higher than a threshold t. We use the two observations to model MAXITER$(b) = \lfloor kb \rfloor$ for some constant k. Then we can compute the total execution time:

$$T_{total} = \text{MAXITER}(b_1)T(b_1) + \sum_{i=1}^{n-1}(\text{MAXITER}(b_{i+1}) - \text{MAXITER}(b_i))T(b_{i+1}).$$

Indeed, when we reach the number of iterations $\text{MAXITER}(b_i)$ we change from precision b_i to precision b_{i+1}. This means that until we are before the iteration $\text{MAXITER}(b_1)$, we compute using b_1 bits. Then until we reach the iteration $\text{MAXITER}(b_2)$, we compute using b_2 bits, that is during $\text{MAXITER}(b_2) - \text{MAXITER}(b_1)$ iterations, and so on until we reach the maximum accuracy with b_n bits at iteration $\text{MAXITER}(b_n)$. We can rewrite T_{total} as

$$\begin{aligned} T_{total} &\approx kb_1T(b_1) + \sum_{i=1}^{n-1} k(b_{i+1} - b_i)T(b_{i+1}) \\ &\approx k(b_nT(n) + \sum_{i=1}^{n-1} b_i(T(b_i) - T(b_{i+1})). \end{aligned}$$

By plugging the expression of $T(b)$ into the previous equation and considering the maximum precision we want is $b_{max} = b_n$, we finally get:

$$T_{total} = k\alpha\left(b_n^2 + \sum_{i=1}^{n-1}(b_i^2 - b_ib_{i+1})\right) + kb_{max}c. \quad (1)$$

Let us consider the function $f(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i^2 - \sum_{i=1}^{n-1} x_ix_{i+1}$. Finding the minimum of $f(x_1, \ldots, x_{n-1}, 1)$ will give us the minimum of the execution time.

By simple partial derivation:

$$\frac{\partial f(x_1, \ldots, x_n)}{\partial x_i} = 2x_i - (1 - \delta_{i,n})x_{i+1} - (1 - \delta_{i,1})x_{i-1}$$

where $\delta_{a,b}$ is equal to 1 if and only if $a = b$, 0 otherwise.

This is where the boundary condition $x_n = 1$ is useful: if we do not set it to a number different from 0, the function is minimized for $x_1 = \cdots = x_n = 0$. Applying this to our problem makes no sense, as we would not be doing any computation. The boundary condition represents the fact that we need to reach an existing precision (b_{max}) eventually. By a simple scaling, considering 1 instead of b_{max} makes computation easier and does not change the resolution of the the following system (up to a factor):

$$\begin{cases} \frac{\partial f(x_1,\ldots,x_n)}{\partial x_1} = 2x_1 - x_2 = 0 \\ \frac{\partial f(x_1,\ldots,x_n)}{\partial x_2} = -x_1 + 2x_2 - x_3 = 0 \\ \vdots \\ \frac{\partial f(x_1,\ldots,x_n)}{\partial x_{n-1}} = -x_{n-2} + 2x_{n-1} - x_n = 0 \\ x_n = 1 \end{cases}$$

Solving this system of equations is equivalent to solving the linear system $Ax = b$ where

$$A = \begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix} \text{ and } b = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

This system has a unique solution which is $\begin{bmatrix} \frac{1}{n} & \ldots & \frac{n-1}{n} \end{bmatrix}$. The minimum of $f(x_1, \ldots, x_n)$ with boundary condition $x_n = 1$ is thus reached for $x_i = \frac{i}{n}$. It is, when multiplied by b_{max}, the solution that minimizes our total execution time.

□

This proof holds only if the execution time evolves linearly with b, but *we could also apply it to minimize the energy consumption* assuming that the energy consumption of one cycle increases linearly with b. In the next section we investigate whether this assumption holds.

C. Evaluation

In order to estimate the cost of our algorithm, we want to model the time of an iteration at precision b. We model an iteration by the following formula: $an^3b^\alpha + c$, where a, c and α are constants, n is the size of the problem and b the number of bits. Indeed, we expect the time to be proportional to the cube of the problem size as we deal with 3D problems, and the parameter α will characterize how the time evolves when we double the number of bits: if $\alpha = 1$, then multiplying by two the number of bits will multiply by two the cost of an iteration (i.e., linear proportion). Maybe the reality could be described by a more complicated polynomial in b, but we want to keep the model simple enough and see if the data can fit it.

To provide numerical values for a, c and most importantly α, we measure the execution times of different scenarios. Each scenario computes 50 iterations on matrices with different sizes, and using either only single-precision floating point variables or only double-precision floating point variables. We denote by $x_{b,n}$ the empirical value obtained using b bits and a problem of size n (i.e. the matrix considered will be of size $n^3 \times n^3$ as we consider 3D problems).

Then, using Python's lmfit package, we interpolate the data to find good values for a, c and α. We are able to estimate different values of α, all between 0.20 and 0.32, using 3 different applications, 2 types of cycles (the classic V-cycle and the *Up* strategy) and 2 types of relaxation methods (weighted Jacobi and an hybrid method). Each data-fitting was done on either 30 or 40 points. With these values of α, we can estimate the cost of our algorithm in units of time by $\left(\frac{b}{53}\right)^\alpha$ for a cycle (1 unit = 1 V-cycle at double-precision) with b the number of significant bits (i.e. the number of bits in the mantissa plus one, as one bit is always assumed in standard floating point representation). Then using the MPFR library we can create different scenarios that set the number of bits used at each cycle in a different way. We define all these scenarios by the first precision (b in the mantissa) used and how to update it

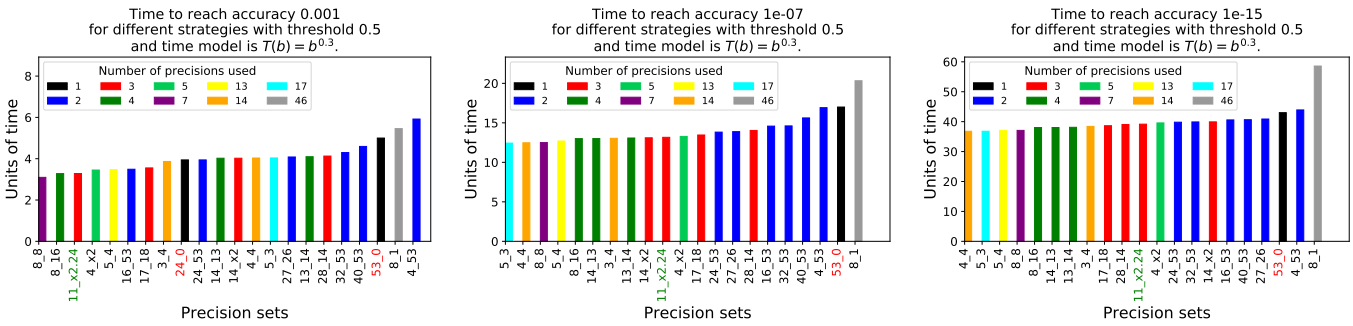


Fig. 8: Cost of the MG solver considering several different dynamic precision scenarios to reach different error degrees in the output: 10^{-3} on the left, 10^{-7} in the middle and 10^{-15} on the right. Red labels indicate current performances with either single-precision or double-precision floating-points. Green label indicates the performance using a mix of half-, single- and double-precision floating point as available on a GPU.

Tolerance	Baseline (DP)	*Up*-cycle (DP)	Adaptive (V-cycle)	Adaptive (*Up*-cycle)	Improvement (DP)	Improvement (SP)
1e-01	1.000	1.333	0.624	0.832	16.8%	-5.5%
1e-02	3.000	2.000	1.872	1.664	44.5%	29.7%
1e-03	5.000	4.667	3.284	3.131	37.4%	20.6%
1e-04	8.000	7.333	5.650	5.234	34.6%	17.0%
1e-05	11.000	10.0	8.015	7.336	33.3%	15.4%
1e-06	14.000	12.667	10.380	9.439	32.6%	14.5%
1e-07	17.000	15.333	13.169	11.964	29.6%	-
1e-08	20.000	18.000	16.169	14.631	26.8%	-
1e-09	23.000	20.667	19.169	17.298	24.8%	-
1e-10	26.000	24.000	22.169	20.631	20.7%	-
1e-11	29.000	26.667	25.169	23.298	19.7%	-
1e-12	32.000	29.333	28.169	25.964	18.9%	-
1e-13	35.000	32.000	31.169	28.631	18.2%	-
1e-14	38.000	34.667	34.169	31.298	17.6%	-
1e-15	43.000	39.333	39.169	35.964	16.4%	-

TABLE III: All estimated times for UNSTRUCTURED-ANISOTROPY, size 40, hybrid relaxation method, on a single processor, with $\alpha = 0.3$. The column 'Improvement (DP)' corresponds to the improvement between the adaptive algorithm using a *Up*-cycle (column 5) compared to the original V-cycle with fixed double-precision (column 2). The column 'Improvement (SP)' corresponds to the improvement between the adaptive algorithm using a *Up*-cycle (column 5) compared to the original V-cycle with fixed single-precision.

($delta$ bits added in the mantissa when the threshold is reached for a given precision). Strategies are denoted as b_delta. This can be either an addition or a multiplication. In particular, we provide a scenario where the available mantissa precisions are 11, 24 and 53 (represented by a starting mantissa precision of 11 and a multiplicator of 2.24, in green on Figure 8). This corresponds to the case where the computation starts in half-precision, then switches to single-precision and finally to double-precision. This scenario is particularly relevant because those precisions are already available in architectures such as the recent Volta GPUs which integrates Tensor cores in half-precision.

Figure 8 represents the cost of the MG solver, for these different scenarios, to reach different accuracy degrees in the output (10^{-3} on the left, 10^{-7} in the middle and 10^{-15} on the right) using a value for the α parameter of 0.3. We can see that the strategy that increases by only 1 the number of mantissa bits (labelled `8_1`) takes more time than the original algorithm (labelled `53_0`) for the 3 different accuracies presented. This is because, in the previous experiments we saw there is usually the same number of cycles in all our strategies (see Figure 7). Here, the convergence rate is limited by slowly increasing number of precision bits: by adding one bit we can reduce by at most 2 the error on the residual norm, while the original algorithm can reduce the error by more than that in one cycle when it is not limited by the bit-width of the variables.

If we focus on a case starting with half precision, then going to single and finally to double precision (labelled `11_x2.24`), we can see that, compared to the original algorithm (which uses only double-precision), we improve by 35% the time to reach accuracy 10^{-3} (see Figure 8). If one adapts the algorithm to use only single-precision variables (labelled `24_0`) as it would be sufficient to reach this accuracy, we would still reduce the time by 17% (still Figure 8). When higher accuracy is needed in the final result, the performance improvement by adapting precisions is still quite important, around 10% faster

than using double precision (See 10^{-15} on Figure 8).

D. Putting it all Together

Finally, we want to see how changing the precisions during the algorithm also improves the execution time when we use the *Up* strategy, defined in section III-D, compared to using the original algorithm (single or double precision). The table III presents the different estimated times (in time units where 1 time unit is a V-cycle at double precision) for all 4 different algorithms: V-cycle (double precision), *Up*-cycle (double precision), V-cycle (adaptive precisions 11-24-53), *Up*-cycle (adaptive precisions 11-24-53) for Problem 1, using a hybrid relaxation method. We also present the improvement of the best method (i.e. *Up*-cycle with different precisions) compared to the original algorithm (in double-precision and in single-precision when possible).

The main result is that even to reach the maximum precision, we improve the execution time by more than 15%. When it comes to smaller precisions, this improvement can be as high as 30% compared to a single-precision algorithm and it even goes up to 45% compared to the original double-precision version. We want to highlight that these results for the adaptive algorithm use 3 different mantissa precisions (11,24,53) (i.e., half, single, double) already available in current hardware. More aggressive adaptations (e.g., `4_4`) could be potentially implemented in architectures such as FPGAs leading to even faster executions, as shown in Figure 8.

V. Related Work

Approximate computing proposals exploit in many different ways the capacity of some computer programs to deliver correct results while operating at low accuracy scenarios: Precision scaling consists in reducing the precision of inputs or intermediate values to reduce storage or computing operands [13], [14]. Other techniques consist in skipping some loop iterations to reduce computational load [15]. Such approaches are reported to provide performance improvements over a factor of two while impacting the final application's accuracy by less than 10%. Load Value Approximation (LVA) consists in exploiting the redundant nature of some computer codes to predict load values and let the execution to progess without stalling [16]. Memoization, which is an approach consisting in storing function results to predict the result of subsequent computations, it is being used to achieve performance enhancements and energy consumption reductions [17], [18]. Fast but slightly incorrect hardware adders have also been explored for video and image compression algorithms [19]. Post-layout simulations show improvements of up to 60% in terms of power and area savings of up to 37% without significant output quality loss. Voltage-scaling techniques have also been explored to reduce energy consumption at the SRAM level, which increases the probability of bit flips [20]. Changing the bit-width of some variables has been proposed [21] to trade image quality for bandwidth.

One similar technique has been proposed to accelerate MG algorithms. It is a strategy consisting in carrying out some relaxation steps while going down the V-cycle and interpolating them to finer grid levels, which has been proposed in [22]. It is aimed at damping perturbations in the linear system solution, even if direct interpolation introduces some degree of error.

VI. Conclusion

This paper improves the original MG algorithm through two distinct approaches: The first one is to remove some relaxation steps in a V-cycle, which leads to faster cycles although requires more of them to converge. Overall, this solution achieves up to 30% improvement although its performance enhancements vary a lot depending on the workload, The second way to improve the algorithm is to adapt the precisions of the floating point depending on MG's precision requirements, We use low precisions levels during the very first MG's cycles and we increase them as the execution progresses. Repeating this operation until reaching the maximum available accuracy available leads to significant speedups. We estimate the benefits of these two approaches combined on different scenarios. When combining half-, single- and double-precisions, we can reduce by 16.4% and 14.5% the execution time compared to using double- or single-precision, respectively, during the whole execution.

In the future we plan to explore additional ideas to further reduce the execution time such as changing the precision used in different levels of a cycle. Also, we plan to estimate the energy efficiency of our technique by evaluating the energy consumption using different values of the α parameter.

VII. Acknowledgements

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 708566 (DURO). The European Commission is not liable for any use that might be made of the information contained therein. This work has been supported by the Spanish Government (Severo Ochoa grant SEV2015-0493)

References

[1] W. Hackbusch, *Multi-Grid Algorithms*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 133–160. [Online]. Available: https://doi.org/10.1007/978-3-642-76717-3_6

[2] S. F. Ashby and R. D. Falgout, "A parallel multigrid preconditioned conjugate gradient algorithm for groundwater flow simulations," *Nuclear Science and Engineering*, vol. 124, no. 1, pp. 145–159, 1996. [Online]. Available: http://dx.doi.org/10.13182/NSE96-A24230

[3] H. Gahvari, A. H. Baker, M. Schulz, U. M. Yang, K. E. Jordan, and W. Gropp, "Modeling the performance of an algebraic multigrid cycle on hpc platforms," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '11. New York, NY, USA: ACM, 2011, pp. 172–181. [Online]. Available: http://doi.acm.org/10.1145/1995896.1995924

[4] M. Casas, B. R. de Supinski, G. Bronevetsky, and M. Schulz, "Fault resilience of the algebraic multi-grid solver," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: ACM, 2012, pp. 91–100. [Online]. Available: http://doi.acm.org/10.1145/2304576.2304590

[5] V. E. Henson and U. M. Yang, "Boomeramg: a parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, vol. 41, no. 5, pp. 155–177, 2002.

[6] R. D. Falgout and U. M. Yang, *hypre: A Library of High Performance Preconditioners*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 632–641. [Online]. Available: http://dx.doi.org/10.1007/3-540-47789-6_66

[7] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 62:1–62:33, Mar. 2016. [Online]. Available: http://doi.acm.org/10.1145/2893356

[8] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, May 2013, pp. 1–9.

[9] B. Grigorian and G. Reinman, "Accelerating divergent applications on simd architectures using neural networks," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 1, pp. 2:1–2:23, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2717311

[10] G. Govindu, L. Zhuo, S. Choi, P. Gundala, and V. K. Prasanna, "Area and power performance analysis of a floating-point based application on FPGAs," in *Seventh Annual Workshop on High Performance Embedded Computing*.

[11] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "Mpfr: A multiple-precision binary floating-point library with correct rounding," *ACM Trans. Math. Softw.*, vol. 33, no. 2, Jun. 2007. [Online]. Available: http://doi.acm.org/10.1145/1236463.1236468

[12] "The gnu mpfr library," http://www.mpfr.org.

[13] T. Yeh, P. Faloutsos, M. Ercegovac, S. Patel, and G. Reinman, "The art of deception: Adaptive precision reduction for area efficient physics acceleration," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 394–406. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2007.41

[14] Y. Tian, Q. Zhang, T. Wang, F. Yuan, and Q. Xu, "Approxma: Approximate memory access for dynamic precision scaling," in *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, ser. GLSVLSI '15. New York, NY, USA: ACM, 2015, pp. 337–342. [Online]. Available: http://doi.acm.org/10.1145/2742060.2743759

[15] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 124–134. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025133

[16] J. S. Miguel, M. Badr, and N. E. Jerger, "Load value approximation," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 127–139. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.22

[17] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy memoization for floating-point multimedia applications," *IEEE Transactions on Computers*, vol. 54, pp. 922–927, 2005.

[18] I. Brumar, M. Casas, M. Moreto, M. Valero, and G. S. Sohi, "Atm: Approximate task memoization in the runtime system," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 1140–1150.

[19] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy, "Impact: Imprecise adders for low-power approximate computing," in *Proceedings of the 17th IEEE/ACM International Symposium on Low-power Electronics and Design*, ser. ISLPED '11. Piscataway, NJ, USA: IEEE Press, 2011, pp. 409–414. [Online]. Available: http://dl.acm.org/citation.cfm?id=2016802.2016898

[20] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 164–174. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993518

[21] J. Park, J. H. Choi, and K. Roy, "Dynamic bit-width adaptation in dct: An approach to trade off image quality and computation energy," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 18, no. 5, pp. 787–793, May 2010. [Online]. Available: http://dx.doi.org/10.1109/TVLSI.2009.2016839

[22] A. Jameson, "Solution of the euler equations for two dimensional transonic flow by a multigrid method," *Applied Mathematics and Computation*, vol. 13, no. 3, pp. 327 – 355, 1983. [Online]. Available: http://www.sciencedirect.com/science/article/pii/009630038390019X