Approximating a Multi-Grid Solver

Valentin Le Fèvre
Barcelona Supercomputing Center
Spain
École Normale Supérieure de Lyon
France
Email: valentin.le-fevre@ens-lyon.fr

Leonardo Bautista-Gomez
Barcelona Supercomputing Center
Spain
Email: leonardo.bautista@bsc.es

Marc Casas
Barcelona Supercomputing Center
Spain
Email: marc.casas@bsc.es

***Abstract*—This paper studies trade-offs between accuracy and execution time in a multi-grid solver: BoomerAMG. As a linear system solver is never really exact, it is actually relevant to investigate whether it is possible or not to save time while keeping results that are not too far from their exact value. We first introduce a different cycle shape than the classic V-cycle used in multi-grid solvers. We show that it provides a usually faster convergence rate by running some tests on a distributed system with huge input problems. Then we propose a dynamic bit-width changing algorithm that is able to adapt the execution time of floating-point operations according to the accuracy needed. In particular, we are able to reach the same results as a double-precision floating point algorithm with an expected 15% improvement of the execution time on a GPU.**

I. Introduction

Multi-Grid (MG) solvers are a class of linear methods [6] 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 this methods display, which have been detailedly analyzed and modeled [**?**]. Also, MG solvers have been reported to display more robustness against silent data corruptions than traditional iterative methods deployed over a single grid [**?**], which also implies they behave well under reduced accuracy scenarios.

Multi-Grid algorithms 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 phases: The *Relaxation*, *Restriction* and *Interpolation* phases. The relaxation phase applies a few steps of an iterative solver like Jacobi or Gauss-Seidelat 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 x with the new error information.

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 fineness of the grid. Different parameters, such as the iterative method used at each step or the fineness 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 for multi-grid algorithms. This paper focuses its effort on one of the most popular parallel implementation of a multi-grid solver, the BoomerAMG [7], implemented in the HYPRE library [3]. This paper makes the following contributions beyond the state-of-the-art:

- We evaluate the impact of parameters of the solver such as the shape of cycles in the grid and the number of iterations.
- We describe a particular cycle that is proved to be at least as efficient as the original algorithm.
- Then we investigate how changing the bit-width of some values makes the execution time vary with the maximum accuracy reachable.
- We give an algorithm that dynamically changes the bit-width of the variables to reduce by 15% the execution time needed to reach a same-quality result as the original algorithm (which uses only double-precision floating points) and reducing up to 30% the execution time for smaller accuracies.

II. The multi-grid algorithm

A. Definitions

- A system of equations is represented by the following equation: $Ax = b$, where $A \in \mathcal{M}(\mathbb{R})^{n\times n}$ and $b \in \mathbb{R}^n$ are given and $x \in \mathbb{R}^n$ is the unknown. The *exact* solution of this system will be denoted by $\widetilde{x}$.
- A level is an integer between 1 and L. Level 1 will be called the finest level, and level L will be called the coarsest level.
- The restriction of A (or b or x) to level l will be denoted by A^l (or b^l or x^l). We have $A^1 = A$ (and $b^1 = b, x^1 = x$).

- We define a set of $L-1$ restriction matrices $R_1, \dots, R_{L-1}$ such that $R_l b^l = b^{l+1}$. We also define some prolongation matrices $P_1, \dots, P_{L-1}$ such that $P_l b^{l+1} = b^l$. In other words, we have $P_l = {R_l}^{-1}$ (left inverse) and we build the A^l matrices as follows: $A^{l+1} = R_l A^l P_l$.
- We denote by e^l the error at level l, that is the vector such that $x^l + e^l = \widetilde{x^l}$, that is to say $\widetilde{x^l} - x^l$. We also define the residual at level l, $r^l = b^l - A^l x^l$. As $b^l = A^l \widetilde{x^l}$, we can also write $r^l = A^l e^l$.
- We derive the relative residual norm at any step i in the algorithm by the norm of the residual at this step, $||b^l - A^l x^l_i||$, divided by the norm of the initial residual, $||b^l - A^l x^l_0||$.

  We also define the notion of *tolerance* as an real value between 0 and 1, which is a threshold for stopping an algorithm. In multi-grid algorithms, this threshold will be on the residual norm.
- We call relaxation a step of an iterative method for solving linear systems (such as Jacobi, Gauss-Sneidel, ...). Formally, for a vector $x \in \mathbb{R}^n$, it represents the computation of $x \leftarrow Mx + c$ where $M \in \mathcal{M}(\mathbb{R})^{n \times n}$ and $c \in \mathbb{R}^n$ are defined depending on the method used.

B. The V-cycle

The goal of the algorithm is to improve the efficiency of iterative methods. Indeed, the choice of the starting vector x on which to apply relaxations has consequences on the convergence time of the solver, and depending on the system to solve, the factor of convergence (related to the matrix M) can be close to 1.

Here the idea is to do some relaxations and then correct the value of x by adding to it the corresponding error term. However, this error term cannot be computed easily (otherwise, solving the problem would be done by computing the error term and adding it to x). Multi-grid solvers instead use recursion to compute the error term. The stopping parameter for the recursion will be determined by decreasing the sizes of vectors and matrices (thus loosing some accuracy but saving time). Formally, we can sum up the algorithm as follows:

```
MG($l, x, f, \alpha_1, \alpha_2$):
  • If $l = L$, return $x = A^{L^{-1}} f$ (exact solve);
  • Else:
      1) Relax $x$ $\alpha_1$ times using an iterative method (matrix $A^l$, right hand side $f$);
      2) $r \leftarrow R_l(f - Ax)$;
      3) $y \leftarrow 0$:
      4) MG($l+1, y, r, \alpha_1, \alpha_2$);
      5) $e \leftarrow P_l y$;
      6) $x \leftarrow x + e$;
      7) Relax $x$ $\alpha_2$ times using an iterative method (matrix $A^l$, right hand side $f$);
```

The algorithm is then executed by setting $x^l \leftarrow 0$ and then executing MG$(1, x^l, b^l, \alpha_1, \alpha_2)$.

Then several ways of modifying the algorithm appear:

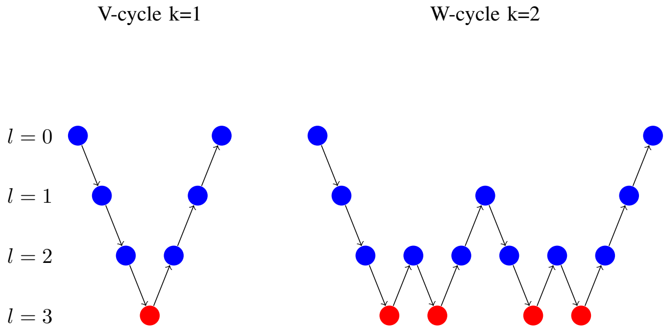


Fig. 1: V-cycle and W-cycle on 4-level grid.

- Which iterative method to use?
- Do we want only one recursive call at each level or more?
- How many times do we need to apply the algorithm?
- How to determine good α_1 and α_2 parameters?
- How many levels should be defined?

In all what follows the iterative method chosen is an hybrid Jacobi/Gauss-Seidel method. The number of levels used will not be studied.

III. Investigate important steps in algorithm

A. Comparison of existing strategies

In this section, we will compare different types of cycles and study how the number of relaxation steps influence the convergence of the algorithm.

We will consider 2 types of cycles: the V-cycle and the W-cycle. The V-cycle is actually the algorithm previously described. The W-cycle looks the same but instead of having two parameters α_1 and α_2 we add a new α_3 parameter. It will represent the number of relaxation steps done after a second recursive call. In terms of algorithm it is the same algorithm as the V-cycle but the bullets 2 to 7 are repeated. We call these cycles V-cycle and W-cycle because of how we can draw them if we represent each time relaxations are done at a level by a point (see Figure 1). It is possible to define other types of cycles by adding more and more repeats of these steps (do k times those steps) 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).

A strategy will be composed of a type of cycle (V or W) and a number of relaxation steps α. The default implementation of BomerAMG does not allow to have different values for $\alpha_1, \alpha_2, \dots$ so we set them all to this value α. We consider a total of 8 different stragies represented in Table I. To compare the different strategies using the BoomerAMG algorithm, we run the algorithm on a predefined matrix of size 512000×512000 for every value of maximum number of iterations (i.e. number of cycles) from 1 to 100 and with a required tolerance of 0 (meaning that the algorithm will stop when the result is exact or the maximum number of iterations is reached). 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. The results are presented on Figure 2.

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

TABLE I: 8 strategies.

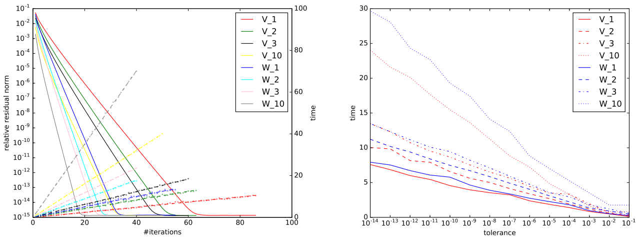


Fig. 2: Execution time and final residual norm of the 8 strategies per iteration (left) and convergence time as a function of the tolerance (right).

What we can observe is that, as expected, increasing the number of relaxation steps or complexifying the cycle increases the overall time to do one cycle. However, it converges in less iterations. 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 increase the complexity of the cycle or do more cycles, thus more moves in the grid, than doing more relaxation steps. This at least proves that multi-grid is a good alternative to classic iterative methods.

B. Improving the baseline

1) Symmetric strategies: Following these results, we can wonder how to improve the efficiency of the simple V-cycle with 1 relaxation step. The first step is to analyze the time spent in the different parts of a cycle. We measure the time spent doing a relaxation at each level and the time spent computing the next linear system (when going down in the grid) or the time spent computing the error term (when going up in the grid). The values where measured for a problem of size 512000 with a 8-level grid and are presented in Table II, along with some information on the matrix used at the corresponding level.

In practice, we notice that the number of non-zero entries in the input matrix is correlated to the average time of a relaxation at a given level. Most importantly, in this example we see that, overall, the relaxation represents $\approx 66\%$ of the total cost of a V-cycle (while the matrix-vector multiplications are only $\approx 30\%$) and that the two first levels are from far the most expensive ones. With this information, there are two ideas: (i) adding more relaxations in the last levels because it is almost free or (ii) removing some relaxations in the first levels to reduce the computational cost.

Level	Matrix size	Non-zero	Relax (down)	Relax (up)	Matvec (down)	Matvec (up)
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 ms	0 ms	0 ms	0 ms
7	69	2,873	0 ms	0 ms	0 ms	0 ms
8	2	4	0 ms	-	-	0 ms

TABLE II: Approximate times spent in the different parts of a V-cycle with $\alpha = 1$.

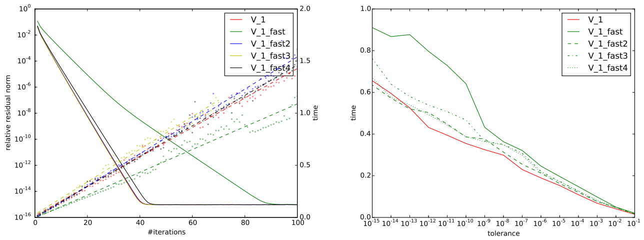


Fig. 3: Execution time and final residual norm for the 4 new strategies on a small matrix.

The following 4 strategies were then tested on this same matrix.

- *Fast* : no relaxation at level 2.
- *Fast2* : 10 relaxations at level $L-2$.
- *Fast3* : 2 relaxations at levels $L-2, L-4, \dots, 3$.
- *Fast4* : no relaxation at level 3.

The strategy *Fast* aims at reducing the cost of the cycle by removing the penultimate relaxation (hoping the accuracy lost at this point will be compensated by the relaxation at level 0) which is very costly. The strategy *Fast4* is a softer version of *Fast* where the relaxation at level 3 is removed, reaching less improvement of the overall execution time but being more easily compensated by the two relaxations at level 1 and 2.
The strategy *Fast2* executes a lot of relaxations at level $L-2$, because it should not increase by much the execution time of the V-cycle. Why choose $L-2$ level instead of L or $L-1$? This because the relaxation at level L is actually a direct solve. Thus, the error term is almost exact at level $L-1$, because the only source of error comes from the interpolation of e^L (which is exact) into e^{L-1}. This is why, we might expect better results by adding relaxations at level $L-2$. The last strategy *Fast3*, pushes this idea one step further. If we assume that doing more than one relaxation gets a really 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 last good estimation of the error vector, hence increasing the number of relaxations again. As we still want not to increase the execution time a lot we stop this recursion for the first levels as they are the two most costly relaxations.

In Figure 3, we present the results of these 4 strategies on a smaller matrix of initial size 64,000 with only a 6-level grid. The first thing to observe is that removing the relaxation at level 1 is a disaster. It sure saves time during a cycle but the accuracy loss is tremendous. The other thing to notice is that adding a lot of relaxations in the last levels increases by a little the execution time while being useless on the accuracy

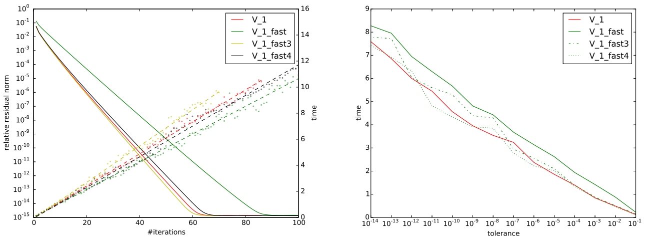


Fig. 4: Execution time and final residual norm for the 4 new strategies on the bigger matrix.

side. This is why *Fast2* and *Fast3* are not really efficient.

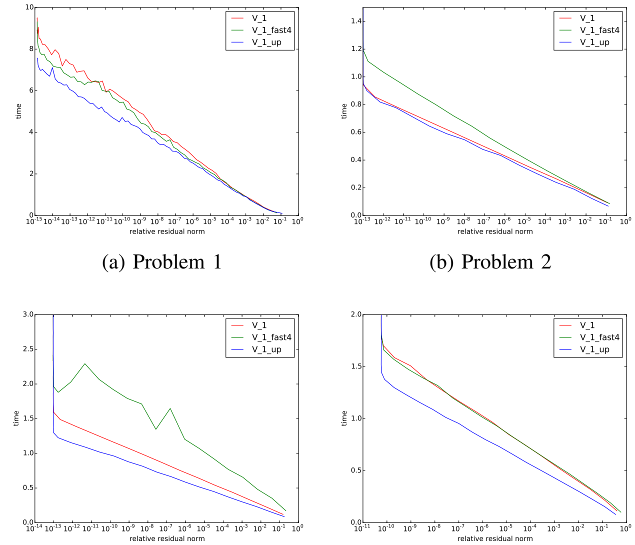
More tests were performed on the original $512,000 \times 512,000$ matrix and are presented in Figure 4.

2) An asymmetric strategy: We observe no big improvement compared to the original V-cycle with 1 relaxation at each level with the previous strategies. Their common point is that they all do the same number of relaxations when going down or up in the cycle. However, the main idea of the multi-grid is totally asymmetric. We first compute an approximation at a given 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. What if we could ensure that any value of the error vectors will be less than ϵ from the exact value just by doing a relaxation step? We would not need to do a relaxation before computing the approximate error term using the next levels in the grid but compute directly the error term when we go back up in the cycle. From that idea we define a new asymmetric strategy: we use a V-cycle with $\alpha_1 = 0$ and $\alpha_2 = 1$. In other words we do one relaxation at each level only when we are going up in the cycle. We call this strategy *Up*.

We run *Up* and *Fast4*, as long as the classical V-cycle, on the same matrix of size 512,000 and also on other applications (3D laplace with a 9-pt stencil, 3D laplace with a 27-point stencil, and another 3D partial derivative equation) with the same size of matrix.

All results are presented in Figure 5.

At this point, we see that it is not worth considering the *Fast4* strategy anymore. However, *Up* seems to be quite efficient since it improves by 12%,7%,20% and 22% (for problems 1,2,3 and 4 respectively) the average time needed to reach a 10^{-i} precision. However all these runs are sequential and does not guarantee the viability of the *Up* strategy when the algorithm is parallelized. More tests were performed on MinoTauro with bigger matrices and different processor topologies. The total size of the matrix is set to either 5,832,000 or 13,824,000, while the topology will be composed of either 27 (3x3x3), 36 (6x6x1) or 64 (4x4x4) processors. The problems tested were the problem 2 and the problem 3 (Laplace equation with either 9 or 27 points stencil). For these 6 possible combinations, we observe an averaged improvement of 18.4% (ranging from 16.0% to 28.3%) for problem 2 and 20.5% (ranging from 16.2% to 25.0%) for problem 3. It seems that *Up* outperforms even more the classical V-cycle when the problem size increases, but seems to cap at around 25% improvement. Figure 6 presents the results for the matrix size 13,824,000 and problem 3, for the 3 different processor topologies. We have similar figures for the other scenarios.



(a) Problem 1

(b) Problem 2

(c) Problem 3

(d) Problem 4

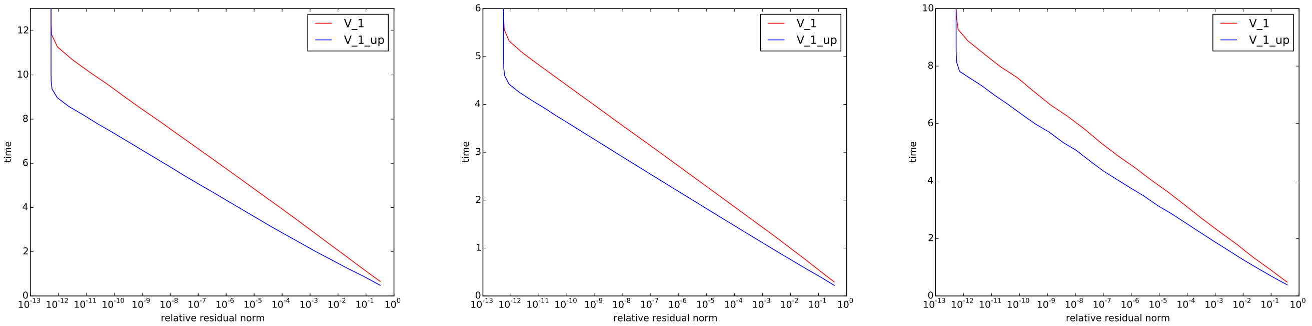
Fig. 5: Comparison of original algorithm with *Fast4* and *Up* strategies.

IV. Investigate Time-Accuracy trade-offs

A. A new version of multi-grid algorithm

If we look at the previous section, we see that we have a lower bound (which depends on the topology and the problem considered) on the accuracy we can achieve. This bound comes from the internal limitations of the `double` type. Indeed, a double uses 8 bytes (52 bits for the mantissa, 11 for the exponent and 1 bit for the sign). This representation does not 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 of the mantissa or the exponent we can describe more and more numbers. In the other way, if we reduce it, we lose accuracy for numbers that are not representable. Considering our algorithm, what if we only want a precision of only 10^{-3} ? We don't need the 64 bits of a `double`, to reach it, a smaller number is enough. The advantage of using less bits for the floating point numbers is a faster and energy-saving computation. Thus it is important to study the impact of using a representation with smaller mantissa.

To realize these experiments, we use two modified versions of the original algorithm. We modify the function that does the relaxation step (remember it is the costly part of the whole algorithm). In this function we can find different internal variables that are originally of type `double`: 5 arrays and 8 scalars. We will denote by *AMG* this original algorithm. The first version, *AMGfloat*, changes the type of these 13



(a) 3x3x3

(b) 4x4x4

(c) 6x6x1

Fig. 6: Comparison of original algorithm with *Up* strategy for Laplace problem with 27-point stencil on a 240x240x240 grid.

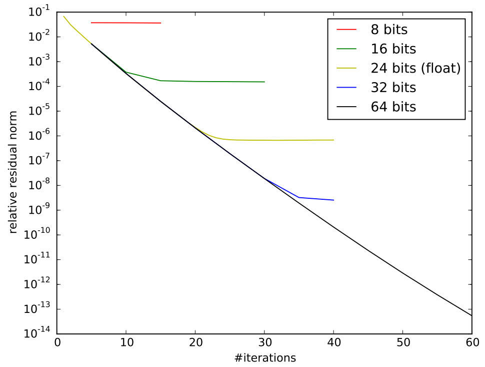


Fig. 7: Accuracy achievable with different number of bits in mantissa. Points (except for 24 bits) were obtained only for multiples of 5 iterations, but it is enough to see the thresholds appear.

variables from `double*` or `double` to `float*` or `float`. For the second version, we use the library MPFR [4], [1] that introduces a type `mpfr_t`. This type has a parameter which is the number of bits in the mantissa of the variable. Every computation is done exactly (assuming the inputs are exact) then rounded to a number using exactly the number of bits set as parameter. We build the second version, *AMGmpfr(b)*, by replacing only the 8 scalar variables by `mpfr_t` variables, all using the same number of b bits for the mantissa. In terms of arithmetic precision, *AMG* behaves similarly to *AMGmpfr(53)* and *AMGfloat* to *AMGmpfr(24)*. There can be small differences due to the rounding method used.

The Figure 7 represent the accuracy we reach using either *AMGmpfr(8)*, *AMGmpfr(16)*, *AMGfloat*, *AMGmpfr(32)* or *AMGmpfr(64)*. The problem is problem 1, with 2x2x2 processor topology and 20x20x20 matrix size. What we actually see on this figure is the threshold reachable depending on the number of bits used. However, until this threshold is reach, the accuracy obtained using any number of bits is the same. This means that, for example, while the accuracy of the solution is below the threshold using 32 bits, using 32 or any greater number of bits to do the computations will result in the same final accuracy. However, we expect that using only the necessary 32 bits is more efficient in terms of time, space and energy.

The second thing to see is that *AMGfloat* behaves exactly as *AMGmpfr(24)* in terms of accuracy whereas more variables were changed from `double` to `float`. It does not degrade more the accuracy, only a few variables from the 8 scalars actually control the final precision as they are temporary variables used for intermediate computations before being plugged back into the input matrix/vector. That being said, we design a new algorithm that adapts the precision of the variables during the execution. It is the same algorithm as *AMGmpfr(b)* except this time the precision can change between two cycles. We fix a threshold on the ratio between the relative residual norm at two consecutive steps to trigger the precision change. Figure 8 presents the evolution of accuracy for the original algorithm, 3 different strategies with increasing precision and a threshold of 0.8: start at $b = 16$ and do $b = b + 8$, start at $b = 32$ and do $b = b + 8$, and start at $b = 16$ and do $b = b \times 2$. We run these strategies on a 240x240x240 matrix size with a 3x3x3 topology for problem 3.

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 by waiting for the ratio between two consecutive relative residual norm to be higher than the threshold, when the precision changes, the gain is more important than that of the original algorithm (i.e. the slope is bigger on the figure), which makes any of the 3 versions able to reach the maximum accuracy (of 4.7×10^{-13}) in the same number of cycles as the original algorithm (20-21 cycles).

Two questions arise at this step. How to evaluate the energy/time savings of this adaptive algorithm? Which precisions should be used to optimize these savings?
The first question is not simple because the use of the MPFR library to do these accuracy experiments introduce a huge overhead in the computations (running *AMGmpfr(53)* is about 20 times longer than running *AMG* whereas it represents the final same algorithm), and this overhead is not highly

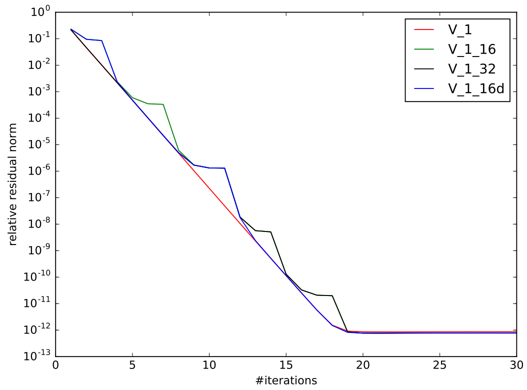


Fig. 8: Accuracy of adaptive algorithms compared to the original double-precision with a threshold for updating the precision of 0.8.

influenced by the choice of the number of bits. Thus it is impossible just to measure the execution time of the algorithm using MFPR library.

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.

Theorem 1. *Given b_{max}, a maximum number of bits, n the number of different precisions $b_1, \dots, b_n$ 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 $b_i = \frac{i}{n} b_{max}$.*

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 previous cycles (Figure 8) and that (ii) the number of cycles needed to reach the lower bound is proportional to the number of bits b used (Figure 7). We then define $\textsc{MaxIter}(b)$ the number of cycles needed so that the ratio between two consecutive relative residual norms is higher than a threshold t. We use the two observations to model $\textsc{MaxIter}(b) = \lfloor kb \rfloor$ for some constant k. Then we can compute the total execution time of the algorithm

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

Indeed, when we reach the number of iterations $\textsc{MaxIter}(b_i)$ we change from precision b_i to precision b_{i+1}. We can rewrite T_{total} as

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

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:

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

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

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

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 $x_1 = \dots = x_n = 0$. This applied to our problem makes no sense, as we would't be doing any computation. The boundary condition represents the fact that we need to reach a 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):

$$\left\{ \begin{array}{rcccccccccc} \frac{\partial f(x_1, \dots, x_n)}{\partial x_1} & = & 2x_1 & - & x_2 & & & & & = & 0 \\ \frac{\partial f(x_1, \dots, x_n)}{\partial x_2} & = & -x_1 & + & 2x_2 & - & x_3 & & & = & 0 \\ & \vdots & & & & & & & & & \\ \frac{\partial f(x_1, \dots, x_n)}{\partial x_{n-1}} & = & & & -x_{n-2} & + & 2x_{n-1} & - & x_n & = & 0 \\ x_n & = & 1 & & & & & & & & \end{array} \right.$$

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 $\begin{bmatrix} \frac{1}{n} & \dots & \frac{n-1}{n} \end{bmatrix}$. The minimum of $f(x_1, \dots, x_n)$ 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 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. The next section will 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: $an^3 b^\alpha + c$, where a, c and α are constants, n is the size of the problem and b the number of bits.

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

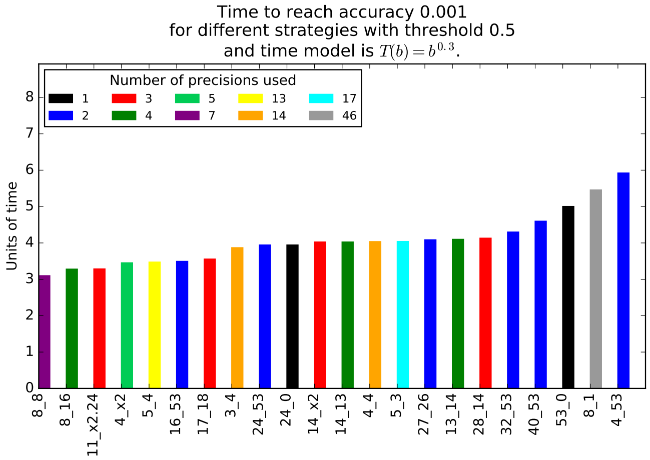


Fig. 9

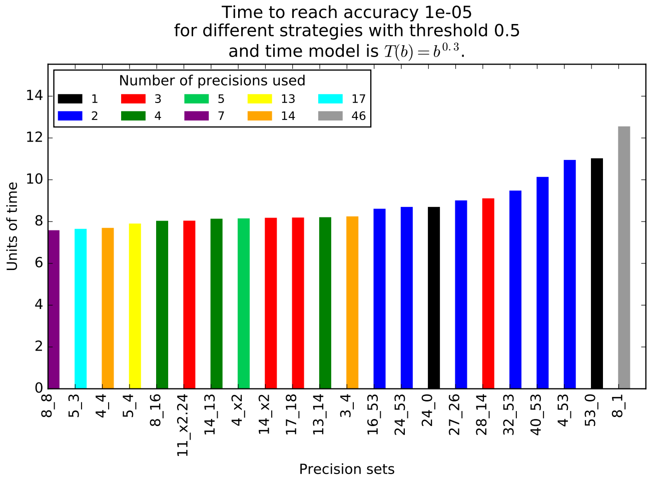


Fig. 10

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 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 method (weighted Jacobi and an hybrid method). 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 used and how to update it (when the threshold is reached for a given precision). This can be either an addition or a multiplication. In particular, we provide a scenario where the available precisions are 11, 24 and 53 (represented by a starting precision of 11 and a multiplicator of 2.24). This corresponds to the case where half-precision, single-precision and double-precision floating points are available, as it is the case on GPUs.

Figures 9 to 13 represents the cost of the algorithm, for these different scenarios, to reach different accuracy in the output (from 10^{-3} to 10^{-15} here) 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`). This is because, in the previous experiments we saw there is usually the same number of cycles in all our strategies (see Figure 8). Here, the convergence rate is limited by the slowly increasing number of precision bits: by adding one bit we can divide by

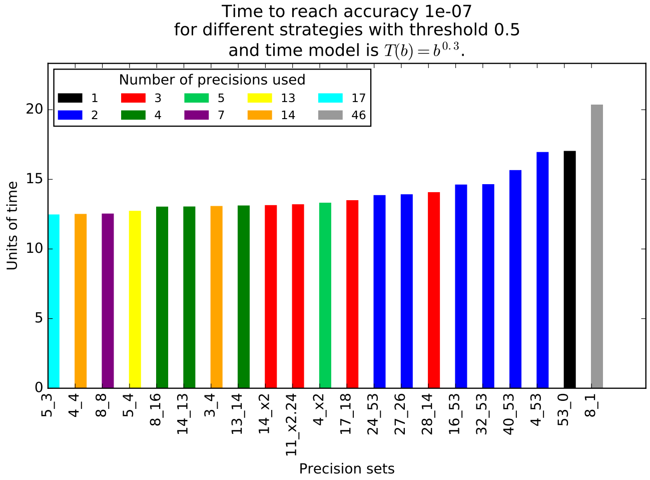


Fig. 11

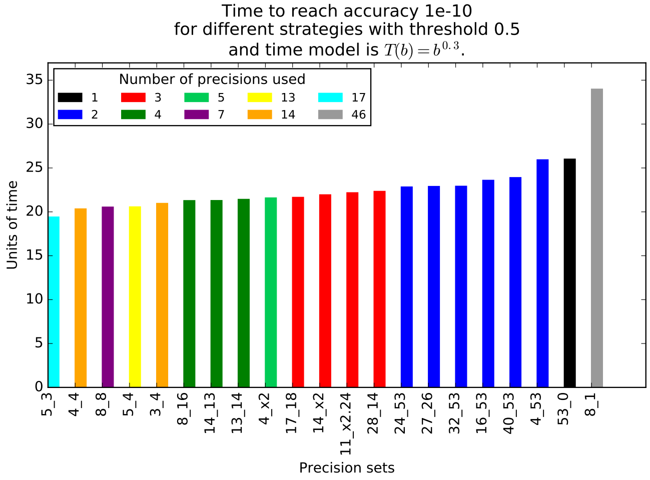


Fig. 12

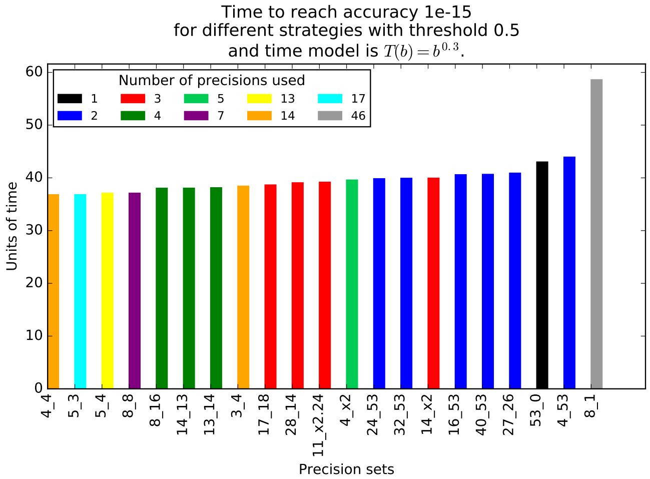


Fig. 13

at most 2 the accuracy on the residual norm while the original algorithm can divide by more than that in one cycle if it is not limited by the bitwidth of the variables. If we focus on the GPU case (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}. 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%. Then the improvement slowly decreases when increasing the accuracy to reach 10%.

D. Summary of results

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-B2, 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 insist to remember that these results for the adaptive algorithm use 3 different precisions (11,24,53) whereas if other architectures would be available it would be possible to improve even more the execution by using more different precisions. For example, as we see on Figure 13, using precisions 4,8,12,16,. . . ,48,52,53 should lead to a faster execution.

V. Related Work

Some approximate computing techniques at hardware level exist and could be applied in addition to our ideas. They include fast but slightly incorrect adders [5], or voltage-scaling (in SRAM, reducing the voltage by 80% increases the probability of an unexpected bit flip by around 10^{-5} [10]) if we are more interested in reducing the energy consumption instead of the execution time.

Concerning multi-grid algorithms, the opposite of our *Up* strategy has been considered in [8]: they do some actual computation when going down in the grid and then simply interpolate to retrieve the solution on the finest grid. The reason for that is that they address a specific problem where some perturbations need to damped quite efficiently, which is why as soon as they perform some computation on the fine grid they need to use coarse grids even if interpolating directly introduces some errors. Dynamically changing the bit-width of some variables has been done in [9] but it is there applied on some part of the data as they are not involved in the same way in the algorithm, which is not the case in a multi-grid solver (all the evaluation points on the grid represent the continuous solution).

VI. Conclusion

To conclude, we improve the original algorithm by two means. The first one is to remove some relaxation steps in a V-cycle. This leads to faster cycles but it converges in more cycles. All in all, it is usually faster (up to 30% according to our experiments) but the improvement can be very variable. The second way to improve the algorithm is to adapt the precisions of the floating point operations depending on which accuracy we already got. We use small precisions for the first cycles then when it starts converging, we increase this precision. Repeating this operation until reaching the maximum accuracy available, leads to some economy in some steps of the algorithm, making it faster overall. Then by combining these two techniques, we estimate than using a GPU, we can reduce by 16.4% the time needed to reach the maximum accuracy using double-precision and by 14.5% the time needed to reach the maximum accuracy using single-precision.

Other ideas to reduce the execution time or increase the convergence could include changing the precision used in different levels of a cycle. However, we think that it is not useful because (1) using a greater precision in the coarse levels than in the fine levels is useless as all the computations would eventually be truncated and (2) using a smaller precision in the coarse levels than in the fine levels would not affect by much the execution time as, according to the measurements done, only the first 2 or 3 levels represent more than 95% of the relaxation cost of a cycle. Finally, it would be interesting to estimate the impact of our algorithm on the energy consumption, or at least have some real measurements and evaluate the energy consumption with our same model but a different value of the α parameter.

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 Problem 1, size 40, hybrid relaxation method, on a single processor, with $\alpha = 0.3$. The column 'Improvement (DP)' corresponds to the improvement between our 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 our adaptive algorithm using a *Up*-cycle (column 5) compared to the original V-cycle with fixed single-precision.

References

[1] The gnu mpfr library. http://www.mpfr.org.

[2] Steven F. Ashby and Robert D. Falgout. A parallel multigrid preconditioned conjugate gradient algorithm for groundwater flow simulations. *Nuclear Science and Engineering*, 124(1):145–159, 1996.

[3] Robert D. Falgout and Ulrike Meier Yang. *hypre: A Library of High Performance Preconditioners*, pages 632–641. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[4] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. Mpfr: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2), June 2007.

[5] Vaibhav Gupta, Debabrata Mohapatra, Sang Phill Park, Anand Raghunathan, and Kaushik Roy. Impact: Imprecise adders for low-power approximate computing. In *Proceedings of the 17th IEEE/ACM International Symposium on Low-power Electronics and Design*, ISLPED '11, pages 409–414, Piscataway, NJ, USA, 2011. IEEE Press.

[6] W. Hackbusch. *Multi-Grid Algorithms*, pages 133–160. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991.

[7] Van Emden Henson and Ulrike Meier Yang. Boomeramg: a parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41(5):155–177, 2002.

[8] Antony Jameson. Solution of the euler equations for two dimensional transonic flow by a multigrid method. *Applied Mathematics and Computation*, 13(3):327 – 355, 1983.

[9] Jongsun Park, Jung Hwan Choi, and Kaushik Roy. Dynamic bit-width adaptation in dct: An approach to trade off image quality and computation energy. *IEEE Trans. Very Large Scale Integr. Syst.*, 18(5):787–793, May 2010.

[10] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan 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*, PLDI '11, pages 164–174, New York, NY, USA, 2011. ACM.