Reduced precision applicability and trade-offs for SLAM algorithms

Oscar Palomar, Andy Nisbet, John Mawer, Graham Riley, Mikel Lujan
Advanced Processor Technologies (APT)
University of Manchester, UK
Email: oscar.palomar@manchester.ac.uk

***Abstract*—This paper presents a study of the use of half-precision (16-bit) floating point for SLAM algorithms, an emerging computer vision problem. Our experiments show that a mix of 16-bit and 32-bit floating point data is needed for the algorithm to function properly, and a library that facilitates mixing 16- and 32-bit floating point variables is introduced. We present our initial results for the whole benchmark selected, showing that 72% of the floating point operations can be performed on 16-bit data, with limited impact on the accuracy of the results. Then we study in depth one of the kernels of the algorithm and the impact of using 16-bit/32-bit floating point values in different variables used in the kernel.**

***Keywords*-computer vision, slam, reduced precision**

I. Introduction

Computer vision is an increasingly important application domain, and has very attractive applications for mobile systems, e.g. augmented reality. But computer vision algorithms are typically compute intensive and require powerful high-end hardware. Thus, it is not straightforward to make these applications suitable for battery-operated devices with a low power budget, in which we need to balance energy consumption, and performance without sacrificing the quality of the output.

An emerging computer vision problem is *Simultaneous Localisation and Mapping*, or SLAM. SLAM algorithms create a 3D map of the surroundings from video captured with a moving camera, and estimate the position and orientation of the camera (the pose) within the map. There are abundant applications of SLAM, in robotics, drones, augmented reality, etc. This is a very active research area and several SLAM algorithms have been proposed recently: KinnectFusion [2], ORB-SLAM [4], [5], ElasticFusion [3] and LSD-SLAM [6]. There are several significant variants of SLAM: they can be *dense* or *sparse*, depending on how they represent the reconstructed 3D scene. Dense variants keep a three-dimensional array, with one entry per voxel. The memory used by the structure depends on how large the scene is and the voxel size, which also affects the accuracy and quality of the output. Sparse representations keep information only for the parts of the space that are not empty. SLAM may also vary with respect to the sensors used: some algorithms (*stereo*) use the differences between two cameras to determine the distances, while others (*RGB-D*) have a single RGB camera coupled with a depth camera, such as the one present in Microsoft's Kinnect.

SLAM applications provide excellent opportunities for applying approximate computing techniques. They have several characteristics that make them attractive for this purpose:

- It may not be necessary to process all frames in full detail. Typically, the delta between consecutive frames is small, thus the information provided by each one of them is small. The exception would be if the camera movement is very abrupt, but that would most probably make the algorithm fail. Also, some SLAM algorithms (e.g. ORB-SLAM) select a subset of frames (called *keyframes*) that are kept separately and receive further processing. These keyframes are considered to provide a significant amount of information with respect to other keyframes. There is clearly an opportunity to perform the computation of non-key frames in an approximate fashion.
- The input is a video stream; thus, the algorithms typically need to account for noise and other sources of inaccuracy (e.g. the depth camera is not always able to return the depth, it depends on the light conditions and the materials of the surroundings).
- SLAM algorithms have several parameters that affect the accuracy of the result. For example, the resolution (voxel size) of the reconstructed 3D map. Thus, it already includes accuracy vs. performance/memory usage/energy trade-offs.
- Using a synthetic input, such as the popular ICL-NUIM dataset [7], we can precisely measure how much the computed trajectory deviates from the actual trajectory. The Absolute Trajectory Error (ATE) indicates the difference between the known ground truth and the estimated trajectory. Therefore, we have a metric that can be used directly to evaluate the impact of the approximate computing techniques being applied, e.g. reduced precision.

We consider the use of reduced precision, a very promising approach for SLAM. Using narrower data (e.g. 16-bit floating point) we can significantly reduce the memory footprint and reduce the energy spent in both moving floating point values and in floating point unit. In this paper, we use SLAMBench [1] to study the use of reduced precision for

SLAM. In its current release, SLAMBench implements the Kinnect Fusion algorithm. We chose SLAMBench because it represents an important class of SLAM (dense and based on depth cameras). Moreover, it has well structured C++ code, parametrised and with a clean division into kernels. Moreover, it provides a framework for easily capturing performance, power and accuracy (e.g. it provides a tool to measure ATE from the log file that SLAMBench generates). Such features have been exploited for studying design space exploration [8].

This paper contributes an initial study of the use of 16-bit floating point variables in SLAMBench, including an analysis of its impact on the accuracy of the trajectory; it also presents a more detailed description and evaluation of one of the most relevant kernels of SLAMBench; finally, it describes a library that enables experimenting with the use of reduced precision mixed with standard 32/64-bit values. Although previous work has shown the potential of reducing the precision for several applications [9], [10], [11], to the best of our knowledge there is no published work presenting an analysis on the applicability of reduced precision to SLAM.

II. SLAMBench

SLAMBench [1] is a publicly available software framework that enables researchers to study trade-offs in performance, energy and accuracy of dense RGB-D SLAM systems. The code provides implementations in C++, CUDA, OpenMP and OpenCL. By sharing the framework and parts of the code amongst all these implementations, SLAMBench facilitates fair comparison of a wide variety of systems with significantly different power and performance characteristics.

SLAMBench implements the KinnectFusion algorithm [2], which has been cleanly divided into multiple kernels. The algorithm has the following phases, shown in Figure 1:

1) *acquire* provides the next RGB-D frame. In our experiments, this means reading one frame at a time from the input file containing the video sequence.
2) *preprocess* transforms the depth information to meters, and applies a filter to the image to reduce noise.
3) *track* is composed of several kernels. This phase estimates the pose for the new frame by generating a point cloud from the frame and then correlating it with the existing 3D map of the environment. The frame's pixels are transformed into 3D points (forming the point cloud); a kernel computes the distance between points in the new point cloud and in the reconstructed 3D map; all distances are reduced and a finally singular value decomposition is used to solve the system, producing the new pose.
4) *integrate* merges the information from the new point cloud, created by track, into the existing map. This must be skipped if tracking was not successful. This kernel is described in much more detail in Section IV-B.
5) *raycast* recovers the 3D surfaces from the volume for the current pose.
6) *rendering* uses the outcome of raycast to generate an image to visualise the depth information, the result of tracking and the reconstructed volume.

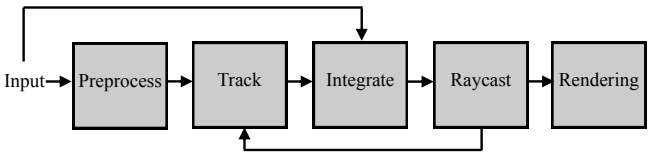

Figure 1. SLAMBench phases

III. FlexiblePoint Library

We developed a C++ library for a flexible floating point representation in which each variable can use 16 or 32 bits. A variable can change its representation (width) at run time, from 16 to 32 bits and vice versa. There is a change in the representation in two cases: 1) the programmer may set explicitly the width of a variable, 2) a 32-bit (or 16) variable is written with the output of an arithmetic operation that produced a 16-bit (32) value. Operations on two variables of the same size generate an output of the same size (i.e. operating on two 16-bit operands results in one 16-bit value, while two 32-bit values generate a 32-bit value). For operations with mixed widths, the current behaviour is to promote the output to 32 bits. However, the logic to decide that is isolated in a function to make it easy to change. The default width is configurable, and it can be changed at run time (e.g. for certain regions of code). The default width is used for new variables, or when casting from other data types.

All necessary operators are overloaded, so simply changing the variable type from `float` to our own `flexpoint` type makes the application code automatically use the correct methods. As a consequence, the majority of the application was successfully converted to `flexpoint` with minor modifications to the code. This was facilitated by the use of explicit conversion operators. However, there were a few exceptions. For example. we had to implement `flexpoint` wrappers for a few library calls (e.g. sin, abs, cos), that have `float`s as input. The compiler was not able to cast from `flexpoint` automatically in these cases.

We added support to compute statistics of the usage of each width. For each arithmetic operation, we count if the 16-bit or the 32-bit representation was used. We can also capture the range of the data stored in the `flexpoint` variables. Finally, we implemented support for dumping the statistics and to pause/resume counting. This last feature enables us to produce statistics only for a given part of the

code. In our case, we use this to analyse a specific kernel of SLAMBench in Section IV-B.

Currently, the library assumes that there is hardware support for 16-bit floating point variables (e.g. ARM, via the `__fp16` data type offered in gcc). The code could be changed to use another representation rather than `__fp16`, to avoid relying on hardware support (or slow dynamic binary translation). One current limitation of gcc's support for half precision variables is that they cannot be function parameters or return values. However, it accepts them as part of a union or struct. This has complicated slightly the code to implement the library.

SLAMBench makes heavy use of the TooN library[1], which implements support for vectors and matrices, including sophisticated operations on these structures (e.g. singular value decomposition). The library is completely templatised and relies on `std::numeric_limits` to determine the implementation of the operations. We had to provide the `std::numeric_limits` for the `__fp16` data type.

IV. Results

A. Full benchmark analysis

This section focuses on the behaviour of the benchmark as a whole. Table I presents the ATE statistics for a few configurations we tested (results are in meters), using trajectory 2 of the ICL-NUIM living room dataset. A preliminary test showed that there is no practical distinction in accuracy between using float or double data types, as we expected. Then, as a first experiment with the FlexiblePoint library we used 16 bits for all variables. The reason was to test an extreme case, in which there is no need for combining 16- and 32-bit types in a flexible way. But this configuration makes the algorithm fail: it is unable to successfully track any frames.

This initial result indicates that at least a few variables of the application require using 32-bit floating point. Our next step was to make 16-bit the default width, and selectively identify these key variables and make them use 32 bits. Using a combination of traces, dumped variables (the reconstructed 3D space) and the debugger to compare against a 32-bit only "gold standard", we attempted to identify which parts of the code went wrong in the 16-bit version, and thus the variables that caused the problem. Although we were able to observe some interesting cases (e.g. subtracted matrices resulting in many 0's rather than in small values, because different values in 32 bits become the same in 16), this approach was not particularly successful, and we still did not manage to make the application track the frames.

For this reason, we changed this approach to the completely opposite one: make 32 bits the default data width, and selectively make variables narrower. This has the obvious advantage of having an initial version that works correctly, that can be iteratively refined to increase the use of 16-bit floating point. If changing a variable to 16 bits stops the program tracking, or impacts the accuracy too severely, it is straightforward to revert the effect, since we know already which variable is responsible for the behaviour.

[1] http://www.edwardrosten.com/cvd/toon/html-user/

configuration	ATE Max	Mean	Total
FP64	0.049	0.020	18.051
FP32	0.049	0.020	18.067
MIXED (72% FP16)	0.452	0.062	54.513

Table I
ATE RESULTS

This has proven a very fruitful path. We incrementally refined the most time-consuming kernels to increase the use of 16-bit floating point. As a result, currently 72% of floating point operations of the whole kernel have 16-bit floating point variables in all their input operands, while the algorithm is still able to track (most of the frames). This obviously has an impact in accuracy, as shown in Table I. We can see that the average ATE increases from 2cm to 6cm, which is still acceptable. However, the maximum and total ATE (accumulated for all frames) have more worrying values.

To understand what happens, we plot the ATE per frame in Figure 2. There is a noticeable **spike** in the ATE between frames 647 and 684. In these frames (but only in these), the algorithm is not able to track the movement of the camera from the video stream. As a consequence, the estimated pose remains unaltered in those frames, increasingly deviating from the actual pose. Finally, this is corrected suddenly when the algorithm is able to track again, calculating a new pose for frames 685 and later. Interestingly, although it is still able to track, ATE for later frames is noticeably higher than for the first part of the video sequence. A possible explanation to this behaviour is that in frame 685 the algorithm is effectively able to find the current position of the camera, but the orientation (not considered in the ATE metric) might be not completely correct. This may result in integrating erroneous information into the volume, which would affect negatively the behaviour of coming frames. A careful examination of the accuracy of the reconstructed volumes will be part of our future work.

The next section discusses in detail one of the kernels of SLAMBench in which we reach a significant percentage of 16-bit floating point operations, but that requires some of its variables to be 32-bit floating point.

B. The Integrate Kernel

The study of the overall benchmark directed our attention to the integrate kernel. As explained in Section II, this kernel integrates the new point cloud generated by previous kernels into the 3D volume that has been constructed from previous frames. Unless otherwise stated, the rest of the kernels of

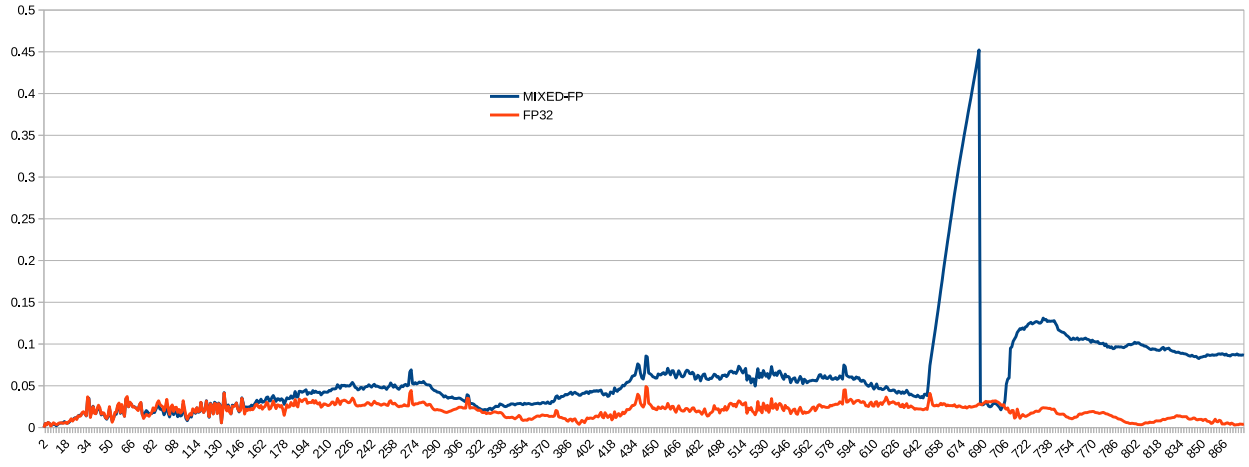

Figure 2. ATE per frame

the benchmark use 32-bit floating point variables. That said, if integrate writes to a variable and sets it into the 16-bit format, this may remain unmodified by other kernels, resulting in limited use of 16-bit computation outside integrate.

Figure 3 lists the source code for this kernel. Its inputs are the volume that represents the reconstructed 3D scene, an array of the depth assigned to each pixel of the image, the (inverted) pose of the camera and several parameters (camera intrinsics, mu and max weight). The output of the kernel is the updated volume data structure. Kinnect Fusion stores the 3D reconstructed volume with a voxel grid. A truncated signed distance function (TSDF) is used to represent 3D surfaces: these are found at zero crossings of the function. The kernel iterates over the volume. Then it uses the camera pose to match voxels with the depth array, and after filtering unusable pixels, the volume is updated. The complete details of the algorithm are described in [2].

We performed an experiment in which integrate uses 32-bit floating point exclusively, while other kernels use 16-bit selectively, just like in the whole benchmark experiments shown in the previous section. The algorithm works correctly, being able to track all frames. As a consequence, the percentage of 16-bit floating point usage for the whole benchmark is quite low (around 45%). Please note that, in addition to the direct usage of 32-bit in integrate, values written in the volume by integrate use 32 bits, and are used unmodified by other kernels, and that the default data width for the whole kernel is still 32: other kernels use 16-bit widths only where explicitly indicated. Changing the default width to 16 bits makes the algorithm unable to track.

A careful study of integrate showed that the data structure storing the camera pose (`cameraX` in Figure 3) is sensitive to the floating point representation used. It is a structure of three floating point values (one for each 3D axes). This variable must use 32-bit floating point data, else the algorithm easily gets lost. Using 16-bit floating point variables in the rest of the kernel works fine as long as `cameraX` uses 32-bit.

Once we identified `cameraX` as a variable that required 32 bits, we went back to use 16 bits as default width for the whole benchmark and change the width to 32 only for that variable. The goal was to determine if this is the only variable that needs 32 bits. This is not the case, since the algorithm is unable to track when using this configuration.

`cameraX` has three components, and we wanted to understand if all three have to use 32 bits. To this end, we ran the benchmark forcing only one of the components of the variable to use 32 bits (the Z axis). A first look at the results is encouraging (see Table II): the benchmark is able to track most of the time (only 41 frames out of 880 are not tracked), but this may be misleading: the average ATE is in the order of dm rather than cm. Moreover, the untracked frames are all found at the end of the sequence. It is possible that a longer sequence would yield worse results. In this experiment, the percentage of 16-bit floating point usage (for integrate alone) rises from 59% to 72%.

Since the interface of integrate and the rest of the kernels is via the volume, we also wanted to see if forcing the members of *vol* that are written to use 32-bit floating point has any effect, while keeping the rest of the benchmark at 16-bit. When forcing `cameraX` and `vol` to use 32 bits, the benchmark reports that it is able to track. Finally, if we use 32 bits for `vol` but 16 for `cameraX`, the behaviour is quite erratic, with the algorithm being able to track a few frames, losing that later, recovering from that, etc.

We captured the ranges of the data stored in the `cameraX` variable, together with `cameraDelta` (which is used to update `cameraX`) and the `vol` variable. Table III lists the results.

```
void integrateKernel(Volume vol, const float* depth, uint2 depthSize,
  const Matrix4 invTrack, const Matrix4 K, const float mu, const float maxweight) {

  float3 delta = rotate(invTrack, make_float3(0, 0, vol.dim.z / vol.size.z));
  const float3 cameraDelta = rotate(K, delta);
  unsigned int y;

  for (y = 0; y < vol.size.y; y++) {
    for (unsigned int x = 0; x < vol.size.x; x++) {
      uint3 pix = make_uint3(x, y, 0); //pix.x = x;pix.y = y;
      float3 pos = invTrack * vol.pos(pix);
      float3 cameraX = K * pos;
      for (pix.z = 0; pix.z < vol.size.z;
        ++pix.z, pos += delta, cameraX += cameraDelta) {
        if (pos.z < 0.0001f) continue; // some near plane constraint
        const float2 pixel = make_float2(cameraX.x / cameraX.z + 0.5f,
          cameraX.y / cameraX.z + 0.5f);
        if (pixel.x < 0 || pixel.x > depthSize.x - 1 || pixel.y < 0
            || pixel.y > depthSize.y - 1) continue;
        const uint2 px = make_uint2(pixel.x, pixel.y);
        if (depth[px.x + px.y * depthSize.x] == 0)
          continue;
        const float diff =
          (depth[px.x + px.y * depthSize.x] - cameraX.z)
          * std::sqrt(1 + sq(pos.x / pos.z) + sq(pos.y / pos.z));
        if (diff > -mu) {
          const float sdf = fminf(1.f, diff / mu);
          float2 data = vol[pix];
          data.x = clamp((data.y * data.x + sdf) / (data.y + 1), -1.f,
            1.f);
          data.y = fminf(data.y + 1, maxweight);
          vol.set(pix, data);
        }
      }
    }
  }
}
```

Figure 3. Source code of the Integrate Kernel

configuration	ATE Max	Mean	Total	Untracked
FP16	3.505	1.904	1675.151	708
VOL-FP32	3.101	1.397	1229.031	385
CAM-Z-FP32	0.712	0.346	304.317	41
CAM-FP32	0.057	0.026	23.164	0
CAM-VOL-FP32	0.056	0.026	23.175	0
FP32	0.049	0.020	18.067	0

Table II
ATE RESULTS FOR SEVERAL CONFIGURATIONS OF THE INTEGRATE KERNEL. THE REST OF THE BENCHMARK USES 32-BIT VARIABLES.

variable	Min	Max	Smallest +	Smallest -
camDelta	-5.417	3.012	1.296e-05	-1.0346e-04
cameraX	-900.791	1433.330	9.313e-10	-3.929e-10
vol	-32736.008	32752.000	1.249e-01	-2.499e-01

Table III
RANGE OF VALUES FOR THE SEVERAL VARIABLES IN THE INTEGRATE KERNEL.

V. CONCLUSIONS AND FUTURE WORK

In this paper we presented initial results that show the potential of using reduced precision in SLAM algorithms. We showed that 72% of the floating point operations could use 16-bit floating point values while still providing acceptable accuracy. However, the algorithm requires using conventional 32-bit floating point variables in parts of the

code. We introduced a library that provides flexible support for mixing 16- and 32-bit floating point variables and changing the representation dynamically.

We plan to focus on the following items as future work:

- *Performance and power evaluation*: We will complete the study presented above with measurement/estimation of the performance and power savings achieved by using reduced precision. We will consider using several hardware implementations that support half precision: ARM CPUs, CUDA GPUs or FPGAs.
- *Automatic width tuning*: The FlexiblePoint library provides mechanisms to set up individual widths for each variable, and to change that dynamically. SLAMBench, when used with an input with a known ground truth, provides a means to measure how well it is doing, the ATE metric. In the experiments presented in this paper, we manually add the library calls to change the widths of the variables and then we observe the impact on the results, in a trial and error fashion. We have not exhausted all the possibilities of the incremental, iterative process to set more variables to use 16-bit floating point. We see a lot of potential in automating this process and performing computer-led design space exploration (DSE) of the widths of the floating point variables.
- *Dynamic tuning*: In this paper sometimes we dynamically set the widths of the variables in different kernels. We have not explored yet the possibility of changing the width according to the progress of the algorithm. For example, we could resort to 32 bits everywhere if we are not able to track successfully.
- *Additional formats*: The FlexiblePoint library currently supports 32-bit and 16-bit floating point types, but it could be easily changed to use different data types. We will consider replacing floating point variables with fixed point representation, where the analysis of the ranges indicate that it is worth doing so.
- *Other SLAM implementations*: We will extend the analysis presented here with studies of alternative SLAM algorithms with significantly different characteristics (e.g. that use a sparse representation and/or stereo cameras).

ACKNOWLEDGMENT

We acknowledge funding by the EPSRC grant PAMELA EP/K008730/1. Palomar is funded by a Royal Society Newton International Fellowship. Luján is funded by a Royal Society University Research Fellowship.

REFERENCES

[1] Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM, L. Nardi, B, Bodin, Z. Zeeshan, J. Mawer, A. Nisbet, P. Kelly, A. Davison, M. Luján, M. O'Boyle, G. Riley, N. Topham and S. Furber. In IEEE International Conference on Robotics and Automation(ICRA), 2015.

[2] KinectFusion: Real-time dense surface mapping and tracking, R.A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohli, J. Shotton, S. Hodges and A. Fitzgibbon. In 10th IEEE international Symposium on Mixed and augmented reality (ISMAR),2011.

[3] ElasticFusion: Dense SLAM Without A Pose Graph, T. Whelan, S. Leutenegger, R.F. Salas-Moreno, B. Glocker and A. J. Davison. In Robotics: Science and Systems (RSS), 2015.

[4] ORB-SLAM: A Versatile and Accurate Monocular SLAM System. R. Mur-Artal, J. M. M. Montiel and J. D. Tards. IEEE Transactions on Robotics, vol. 31, no. 5, pp. 1147-1163, 2015.

[5] ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras, R. Mur-Artal and J. D. Tardos. ArXiv preprint arXiv:1610.06475.

[6] LSD-SLAM: Large-Scale Direct Monocular SLAM, J. Engel and T. Schöps and D. Cremers. In European Conference on Computer Vision (ECCV), 2014.

[7] A Benchmark for RGB-D Visual Odometry, 3D Reconstruction and SLAM, A. Handa, T. Whelan, J. McDonald and A. Davison. In IEEE International Conference on Robotics and Automation (ICRA), 2014.

[8] Integrating algorithmic parameters into benchmarking and design space exploration in dense 3D scene understanding, B. Bodin, L. Nardi, Z. Zeeshan, H. Wagstaff, G. S. Shenoy, M. Emani, J. Mawer, C. Kotselidis, A. Nisbet, M. Lujan, B. Franke, P. H. J. Kelly, and M. OBoyle. In International Conference on Parallel Architectures and Compilation Techniques (PACT), 2016.

[9] EnerJ: Approximate data types for safe and general low-power computation, A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze and D. Grossman. In ACM SIGPLAN Notices, vol. 46, no. 6, pp. 164-174, 2011.

[10] Reducing power by optimizing the necessary precision/range of floating-point arithmetic, J. Y. F. Tong, D. Nagle and R. A. Rutenbar. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 8 no. 3 pp. 273-286, 2000.

[11] The use of imprecise processing to improve accuracy in weather & climate prediction, P. D. Düben, H. McNamara and T. N. Palmer. Journal of Computational Physics, vol. 271, pp. 2-18, 2014.