Memory Behavior of the UAV Application

Grigorios Magklis

Semester project for CSC-573

1 Introduction

In this project we studied the memory behavior of the Unmanned Airborne Vehicle (UAV) application. UAV is a key application for the Complexity Adaptive Processing project (`http://www.ccs.rochester.edu/projects/cap`). The application is using little memory, compared to others, but it is designed to run on an embedded processor. Embedded processors usually have less resources than high-end microprocessors and we would like to know what is the smallest cache that can accomodate most of the application's needs. To do this we are using the Reda simulator to gather statistics about the reuse distance pattern of the application and estimate the best cache size.

2 The application

The application is basically a target recognition system. In its typical environment it periodically receives an image from a frame buffer and tries to identify targets in the image. The images are 128x128 pixels in size, 1-byte per pixel, to a total of 16KB per image. The application is going through three main computation phases.

The first phase is termed *pre-screening* and is trying to find *possible* targets in the image. In order to do this the application is calculating a convolution of the image with a standard 3x3 kernel. The whole image is accessed during this phase, in a normal, left-to-right, top-to-bottom fashion.

The second phase is called *quick clustering* and is trying to identify a set of targets from a set of possible targets. The possisions of all the possible targets are compared to each other and if they are close they are considered parts of the same target. The position of each target is the gravity center of it's clustered possible targets.

In the third phase, called *compute distance*, the application is calculating the distance from the targets identified in the second phase. This step involves computing three distances, one for each of the X, Y and Z axis. The computation is the same for all three distances and it involves three floating-point arrays, representing complex numbers. The first array is called ROI (Region Of Interest) and it represents the data of the image around a selected target (as complex numbers). The second is an HDCCF filter, and the third is a temporary array called BUF. The computation steps in this phase are the following:

1. Compute the 2-dimensional Fast Fourier Transform (FFT) of ROI.
2. Do an element-by-element multiplication of ROI and HDCCF and calculate the surface power of the result.
3. Do an element-by-elemnt multiplication of ROI and HDCCF and put the result in BUF.
4. Compute the 2-dimensional Inverse Fast Fourier Transform (IFFT) of BUF.
5. Do an element-by-elemnt multiplication of ROI and HDCCF and put the result in BUF.
6. Compute the 2-dimensional Inverse Fast Fourier Transform (IFFT) of BUF.

The third phase is repeated for every target in the image, and the whole process is repeated for every image.

3 The source code

The application is written in the C programming language and the code is split in 25 source files and 4 header files. From the 25 files we instrumented only 18. The total code size is 1232 lines, and from that we instrumented 876 lines. The result of the instrumentation was about 4.8 thousand lines.

The code was modified from its original version. Some of the arrays, used in the FTT and IFFT code, were arrays of pointers, and some two-dimensional arrays. The Reda library was not working with theses types of arrays so we converted them to one-dimensional arrays. Part of this modification involved changing the arrays from being dynamically allocated, via `malloc`, to statically allocated global variables. The resulted code has

the following global arrays: (a) three floating-point arrays of 8K, 2K and 2K elements respectively, (b) three integer arrays of 4M, 16K and 1K elements, and (c) a byte (unsigned char) array of 4M elements. The two 4M element arrays are artificial, and they correspond to 256 times the size of an image (16K elements). They could both be replaced with an array of 16K elements that is reused in every iteration.

4 Results

We run the application with 5 different input sizes to see how the input size affects its reuse distance behavior. The different inputs consist of 1, 2, 4, 8 and 16 images from a sequence of 180 frames taken from a real missile camera. The results for each run were compared with the unmodified application to check the correctness of the instrumented code and the simulator. The unmodified application executes about 3.8M instructions to process each image (this number was taken from simulation with the SimpleScalar tool). The simulator outputs statistics based on the base-type of the arrays in the application. So we got four different numbers: total references, floating-point references, integer references and unsigned char (byte) references.

The first thing we notice is that the number of references for each reuse distance scales linearly with the input size (figure 1). This hints to the fact that the memory behavior of the application should not depend on the input size. Figure 2 shows the percentage of memory references for each reuse distance, and this is the same for all different input sizes. As it is expected this is true for each individual type of array (figures 3, 4 and 5).

The interesting thing to notice is that most of the references are due to accesses to the floating-point arrays. Figure 6 graphically shows the number of references for each type for each reuse distance. Accesses to the byte arrays dominate the 2^4, 2^9 and 2^{15} distances. Accesses to the integer arrays show about 30% of total accesses only for the distance of 2^8. Floating-point accesses are the only accesses for most of the distances.

The last figure (figure 7) shows the percentage of accesses that happen up to a certain distance. This figure can help decide what memory size better fits our needs. As we can see a (fully-associative) cache that can fit 8K (2^{13}) elements can accommodate about 95% of the memory references of the application.

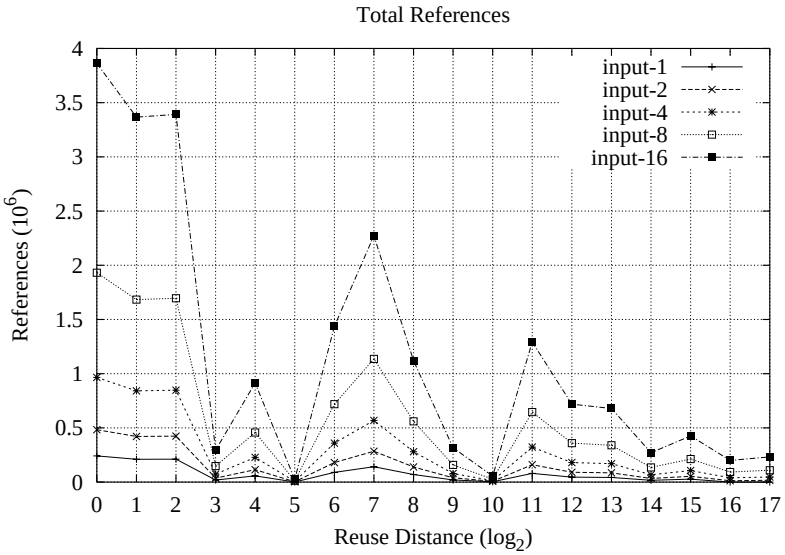

Figure 1: Number of total memory references for each reuse distance, for different input sizes.

5 Conclusions

Through this work we found out that UAV has a standard reuse distance pattern, that does not depend on the input size. We also concluded that the floating-point arrays account for almost all the memory references in the application and that 95% of the memory references have a reuse distance of 2^{13} or less.

Unfortunately there is something that the simulator in its current state cannot tell us. We know, by using different tools, that UAV goes through three distinct phases when analyzing each input image. It would be more helpful, for our purposes, to gather reuse distance information for each phase separately. It would be interesting to see what type of arrays contribute to each phase of the application, and what is the maximum reuse distance for each phase.

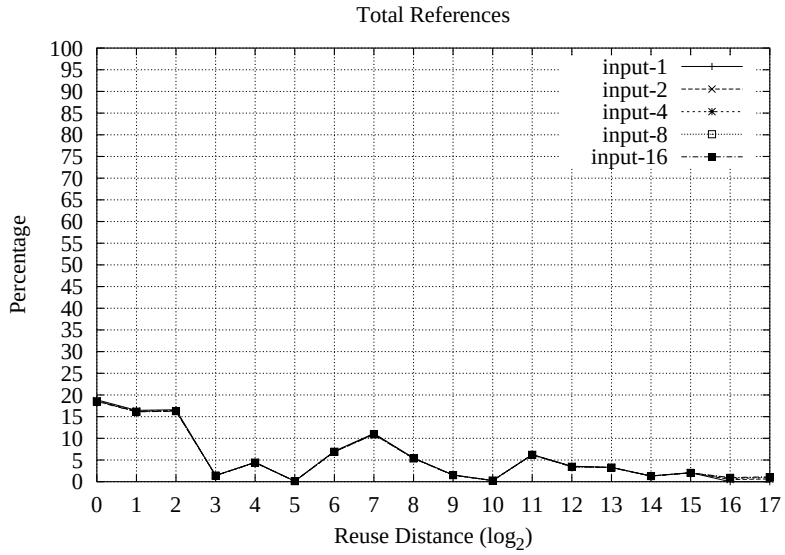

Figure 2: Percentage of total memory references for each reuse distance, for different input sizes.

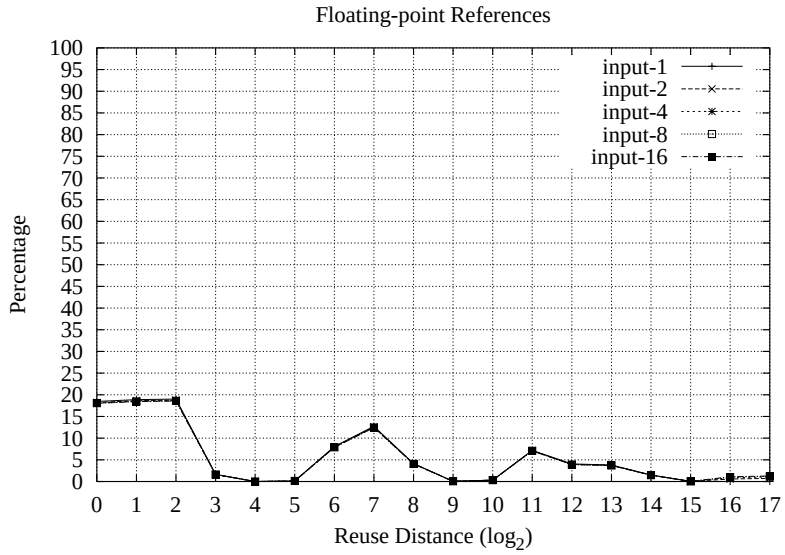

Figure 3: Percentage of floating-point memory references for each reuse distance, for different input sizes.

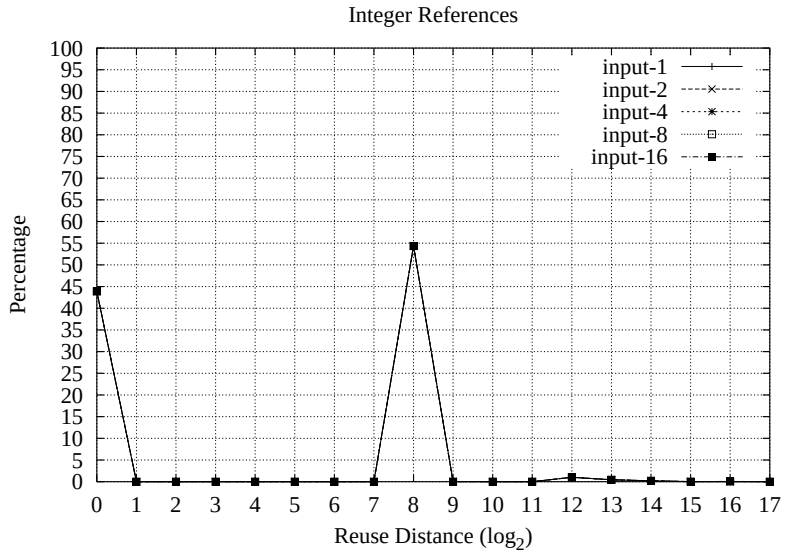

Figure 4: Percentage of integer memory references for each reuse distance, for different input sizes.

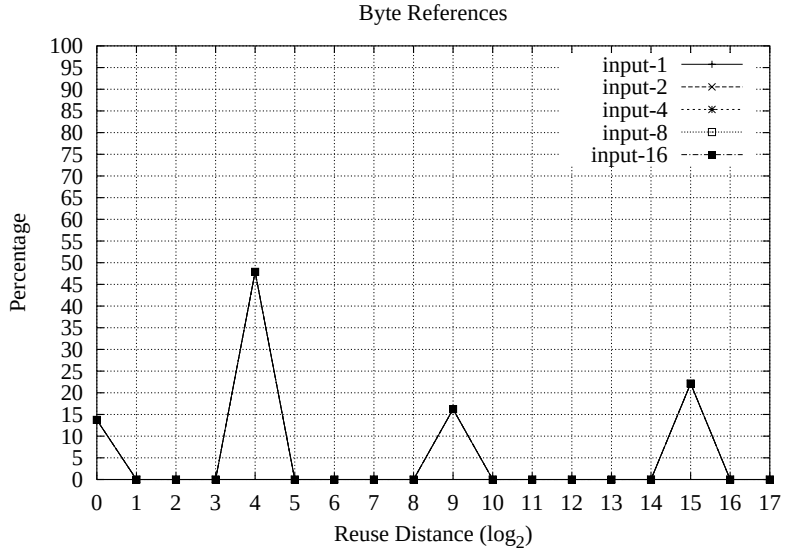

Figure 5: Percentage of byte (unsigned char) memory references for each reuse distance, for different input sizes.

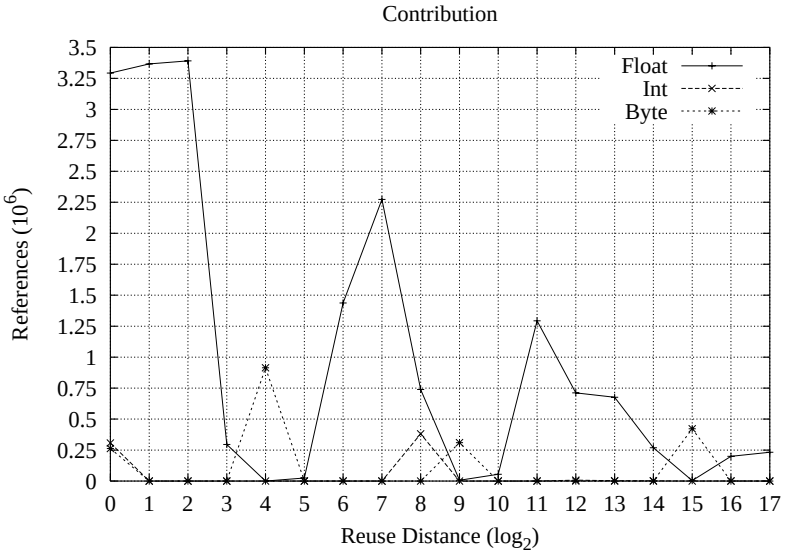

Figure 6: Number of floating-point (Float), integer (Int), and unsigned char (Byte) references, for an input of 16 images.

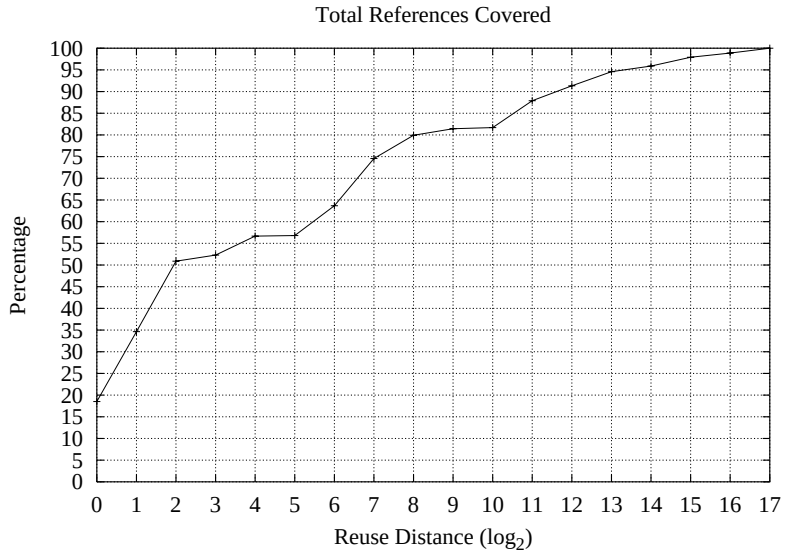

Figure 7: Percentage of memory references covered by each reuse distance, for an input of 16 images.