Implementing the Projected Spatial Rich Features on a GPU

Andrew D. Ker

Oxford University Department of Computer Science, Parks Road, Oxford OX1 3QD, England.

ABSTRACT

The Projected Spatial Rich Model (PSRM) generates powerful steganalysis features, but requires the calculation of tens of thousands of convolutions with image noise residuals. This makes it very slow: the reference implementation takes an impractical 20–30 minutes per 1 megapixel (Mpix) image. We present a case study which first tweaks the definition of the PSRM features, to make them more efficient, and then optimizes an implementation on GPU hardware which exploits their parallelism (whilst avoiding the worst of their sequentiality). Some nonstandard CUDA techniques are used. Even with only a single GPU, the time for feature calculation is reduced by three orders of magnitude, and the detection power is reduced only marginally.

Keywords: Steganalysis, Rich Features, CUDA, GPU programming

1. INTRODUCTION

The modern steganalysis paradigm, particularly for still images, is to apply machine learning techniques to *features* extracted from each image. In early steganalysis[1] the features were small (at most a few hundred per image) and aimed at particular steganographic artifacts. Recently, features have become larger (tens of thousand per image) and more generic. As of 2012, the most accurate steganalysis came from applying a variety of content-suppression filters, quantizing the residuals, and then computing co-occurrence histograms.[2,3] The large dimensionality comes from the enormous number of histogram bins in the high-dimensional co-occurrence matrix. In 2013, a new proposal[4,5] was to apply many random convolutions to the residuals, quantize, and then compute first-order histograms: the *Projected Spatial Rich Model* (PSRM). Each histogram captures a sort of cross-section of the high-dimensional co-occurrence matrix, and the large number of random convolutions ensures that sufficient information is captured. It was shown that PSRM detection accuracy saturates at lower dimension than for features based on co-occurrence histograms.

Early features were almost trivial to compute, but modern features have significant cost. The reference implementation of symmetrized co-occurrence features $\mathcal{CF}^*$[6] takes about 1.3 seconds for a 1Mpix image on our benchmark machine; SRM[2] features take about 12s, and JRM[3] features 36s. Calculation of PSRM features, for a single 1Mpix image, takes 20–30 minutes (see Sec. 5.1 for PSRM benchmarks). The reason is that the PSRM calculation, with default parameters, applies approximately 60 000 convolutions with kernels averaging approximately 20 pixels: of the order of one million floating-point operations *per pixel* of the image analyzed.

We argue that this renders the features almost entirely impractical for both researchers and practitioners. In a hypothetical image forensics situation, hours of analysis for a dozen images is probably too slow an investigation. Training a machine learning algorithm, which will need at least thousands of examples, is very expensive (see Sec. 5.2 for an example of such a dollar cost); researchers, who might easily need to compute features from a million images[7] for testing steganalysis methods, weep at these computational requirements. In a hypothetical online network scanning application, steganalysis taking even one second per image might be too slow.

The author, one of those who wept at the computational costs of PSRM, resolved to create an optimized version. Because multiple convolutions are highly parallelizable, it is natural to turn to a GPU implementation. This paper is a case study describing how to adjust the definition of the features, exploit some symmetries, and then optimize their implementation on a GPU. In this paper, Sec. 2 describes the PSRM features in sufficient detail to understand their computational demands; Sec. 3 briefly outlines the target GPGPU architecture, and its limitations; Sec. 4 describes how we adapted features ('GPU-PSRM') to make them more optimization-friendly, and implemented them; Sec. 5 includes speed benchmarks and a simple steganalysis experiment to check that GPU-PSRM features have comparable power to PSRM, and Sec. 6 concludes the paper.

Further author information: E-mail: adk@cs.ox.ac.uk, Telephone: +44 1865 283530, www.cs.ox.ac.uk/andrew.ker.

$$F_l^1 = \begin{pmatrix} 0 & 0 & 0 \\ +1 & -1 & 0 \\ 0 & 0 & 0 \end{pmatrix},\ F_r^1,\ F_u^1,\ F_d^1 \text{ by rotation},\ F_{lu}^1 = \begin{pmatrix} +1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \end{pmatrix},\quad F_{ru}^1,\ F_{ld}^1,\ F_{rd}^1 \text{ by rotation.}$$

$$F_h^2 = \begin{pmatrix} 0 & 0 & 0 \\ +\frac{1}{2} & -1 & +\frac{1}{2} \\ 0 & 0 & 0 \end{pmatrix},\ F_v^2 = \begin{pmatrix} 0 & +\frac{1}{2} & 0 \\ 0 & -1 & 0 \\ 0 & +\frac{1}{2} & 0 \end{pmatrix},\ F_d^2 = \begin{pmatrix} +\frac{1}{2} & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & +\frac{1}{2} \end{pmatrix},\ F_m^2 = \begin{pmatrix} 0 & 0 & +\frac{1}{2} \\ 0 & -1 & 0 \\ +\frac{1}{2} & 0 & 0 \end{pmatrix}.$$

$$F_l^3 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ -\frac{1}{3} & +1 & -1 & +\frac{1}{3} & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix},\ F_r^3,\ F_u^3,\ F_d^3 \text{ by rotation},\ F_{lu}^3 = \begin{pmatrix} -\frac{1}{3} & 0 & 0 & 0 & 0 \\ 0 & +1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & +\frac{1}{3} & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix},\ F_{ru}^3,\ F_{ld}^3,\ F_{rd}^3 \text{ by rotation.}$$

$$F_l^{3\times3} = \begin{pmatrix} -\frac{1}{4} & +\frac{1}{2} & 0 \\ +\frac{1}{2} & -1 & 0 \\ -\frac{1}{4} & +\frac{1}{2} & 0 \end{pmatrix},\ F_r^{3\times3},\ F_u^{3\times3},\ F_d^{3\times3} \text{ by rotation},\ F_a^{3\times3} = \begin{pmatrix} -\frac{1}{4} & +\frac{1}{2} & -\frac{1}{4} \\ +\frac{1}{2} & -1 & +\frac{1}{2} \\ -\frac{1}{4} & +\frac{1}{2} & -\frac{1}{4} \end{pmatrix}.$$

$$F_l^{5\times5} = \begin{pmatrix} -\frac{1}{12} & +\frac{1}{6} & -\frac{1}{6} & 0 & 0 \\ +\frac{1}{6} & -\frac{1}{2} & +\frac{2}{3} & 0 & 0 \\ -\frac{1}{6} & +\frac{2}{3} & -1 & 0 & 0 \\ +\frac{1}{6} & -\frac{1}{2} & +\frac{2}{3} & 0 & 0 \\ -\frac{1}{12} & +\frac{1}{6} & -\frac{1}{6} & 0 & 0 \end{pmatrix},\ F_r^{5\times5},\ F_u^{5\times5},\ F_d^{5\times5} \text{ by rotation},\ F_a^{5\times5} = \begin{pmatrix} -\frac{1}{12} & +\frac{1}{6} & -\frac{1}{6} & +\frac{1}{6} & -\frac{1}{12} \\ +\frac{1}{6} & -\frac{1}{2} & +\frac{2}{3} & -\frac{1}{2} & +\frac{1}{6} \\ -\frac{1}{6} & +\frac{2}{3} & -1 & +\frac{2}{3} & -\frac{1}{6} \\ +\frac{1}{6} & -\frac{1}{2} & +\frac{2}{3} & -\frac{1}{2} & +\frac{1}{6} \\ -\frac{1}{12} & +\frac{1}{6} & -\frac{1}{6} & +\frac{1}{6} & -\frac{1}{12} \end{pmatrix}.$$

Figure 1. The thirty linear filters used in the first stage of the PSRM calculation.

2. THE PSRM FEATURES

In this section we briefly describe how the PSRM features are calculated. The calculation inputs a bitmap (raw) image I and, implicitly, a set of random kernels, and outputs $234T$ features (originally integers, but see Sec. 5), where T is the number of random projections per residual. Unlike the steganalysis publications that presented the rich models,[2,5] our focus and choice of notation is on *what* needs to be computed, rather than *why*: for implementation, it is the former that we need to understand. The calculation has three stages:

(1) filtering of I by linear and then nonlinear (max/min) operations to create *residuals*;

(2) repeating T times, each residual is convolved with a random kernel, quantized, and the central part of its histogram taken: we call these the *pre-features*;

(3) various sets of pre-features are combined, and finally concatenated to make the *features*.

There is symmetrization (summing of counts that should be equal, assuming various symmetries of natural images) throughout the process. We briefly describe each step in turn.

Step 1 is the simplest. The authors suggest 30 linear filters, which are shown concisely in Fig. 1. There are horizontal, vertical, and diagonal filters of length 2, 3 and 4 (the 's1', 's2', and 's3' models in the reference implementation, called '1st', '2nd' and '3rd' in Ref. 2), as well as 3×3 and 5×5 filters ('s3x3' and 's5x5', called 'EDGE3x3' and 'EDGE5x5' and incorporating 'SQUARE' from Ref. 2). The outputs of these filters are then combined in various (nonlinear) ways by taking pixel-wise maxima and minima; the outputs of these operations are the *residuals*. For example, some of the residuals used in PSRM are

$$R_{rul}^1 = \max(I * F_r^1, I * F_u^1, I * F_l^1), \qquad \bar{R}_{rul}^1 = -\min(I * F_r^1, I * F_u^1, I * F_l^1),$$
$$R_{urd}^1 = \max(I * F_u^1, I * F_r^1, I * F_d^1), \qquad \bar{R}_{urd}^1 = -\min(I * F_u^1, I * F_r^1, I * F_d^1),$$

where $*$ denotes 2-dimensional convolution, and max and min are element-wise. (The notation here is arbitrary and does not follow Refs. 2 or 5.) The minimum residuals (denoted above $\bar{R}$) have their sign flipped, which is part of the symmetrization. In all, a total of 168 residuals are needed: they will not all be listed here.

Step 2 is the computationally demanding part. First, each residual is convolved with a pseudorandom kernel: each kernel has dimension $w \times h$, where w and h are independent and uniformly distributed on $\{1, 2, \ldots, 8\}$; the entries of the kernels are drawn as standard Gaussian, then scaled so that the sum-square of the entries is one.

Then, for each kernel, the histogram of its convolution with the residual is computed, and the central bins counted. The default parameters use six bins, centred on zero, each of unit width. At this stage there is some symmetrization: each kernel is re-used after flipping horizontally, vertically, and both, and the histogram counts totalled. We call these counts *pre-features* (there are six integers for each kernel K and each residual R):

$$\mathrm{PF}(R, K) = \mathbf{H}_{-3}^{+3}\big[R * K\big] + \mathbf{H}_{-3}^{+3}\big[R * K^{\updownarrow}\big] + \mathbf{H}_{-3}^{+3}\big[R * K^{\leftrightarrow}\big] + \mathbf{H}_{-3}^{+3}\big[R * K^{\updownarrow\leftrightarrow}\big], \tag{1}$$

where $\mathbf{H}_i^j$ denotes the $j - i$ histogram bins between integers i and j, $+$ is elementwise, and $\leftrightarrow$ and $\updownarrow$ flip the kernel horizontally and vertically. This process is repeated with many random kernels, $T = 55$ of them with default parameters*, and also with each kernel transposed:

$$\begin{aligned}\mathrm{PF}(R) &= \mathrm{PF}(R, K_1) \cdot \mathrm{PF}(R, K_2) \cdot \cdots \cdot \mathrm{PF}(R, K_T),\\ \mathrm{PF}^t(R) &= \mathrm{PF}(R, K_1^t) \cdot \mathrm{PF}(R, K_2^t) \cdot \cdots \cdot \mathrm{PF}(R, K_T^t),\end{aligned}$$

where $\cdot$ represents concatenation and $-^t$ the matrix transpose. Each pre-feature has $6T$ integer entries.

At step 3, the pre-features are further summed (for symmetrization) in combinations that combine filters of similar sizes. For example, the $6T$ features called 's1_minmax_34h' are computed as

$$\text{'s1\_minmax\_34h'} = \mathrm{PF}(R^1_{rul}) + \mathrm{PF}(R^1_{ldr}) + \mathrm{PF}(\bar{R}^1_{rul}) + \mathrm{PF}(\bar{R}^1_{ldr}) + \mathrm{PF}^t(R^1_{dru}) + \mathrm{PF}^t(R^1_{udl}) + \mathrm{PF}^t(\bar{R}^1_{dru}) + \mathrm{PF}^t(\bar{R}^1_{udl}). \tag{2}$$

The above represents one of 39 *submodels* making up the PSRM features, each of $6T$ features. With $T = 55$ there are a total of 12870 features. Almost all submodels are calculated using formulae similar to (2), but a few have a further degree of symmetrization: when convolved residuals should be symmetric about zero (which is when no max or min operations are used), their *absolute value* is binned and counted: instead of (1) we have

$$\mathrm{PF}(R, K) = \mathbf{H}_{0}^{+3}\big[\mathrm{abs}(R * K)\big] + \cdots$$

However, there are few such submodels, and they can be absorbed into the above framework by adding the negative histogram bins to the positive at the final stage. For example, the $6T$ features called 's1_spam14hv' are computed from residuals $R^1_r = I * F^1_r$ and $R^1_u = I * F^1_u$ as

$$\begin{aligned}\text{'s1\_spam14hv'} = &\Big(\mathrm{PF}(R^1_r)[3,2,1] + \mathrm{PF}(R^1_r)[4,5,6] + \mathrm{PF}^t(R^1_u)[3,2,1] + \mathrm{PF}^t(R^1_u)[4,5,6]\Big) \cdot\\ &\Big(\mathrm{PF}(R^1_u)[3,2,1] + \mathrm{PF}(R^1_u)[4,5,6] + \mathrm{PF}^t(R^1_r)[3,2,1] + \mathrm{PF}^t(R^1_r)[4,5,6]\Big)\end{aligned}$$

where $[3, 2, 1]$ indicates an array slice.

We can now see why PSRM features are so demanding to compute. Suppose that the input image I has n pixels. Step 1 is fairly trivial: convolutions of I with 30 kernels each of maximum size 25 (fewer than $750n$ operations†), and 168 min-max elementwise operations ($168n$). Step 3 is trivial. Step 2 is dominant: for each of T kernels, 168 residuals, and 8 symmetrizations, one computes a convolution of I with a kernel of mean size $\frac{81}{4}$ elements (and then the quantization and bin counting, which takes just a little more work). If $T = 55$, this requires a total of approximately $1.5 \times 10^6 \cdot n$ FLOPs. A small amount is saved by avoiding duplication of convolution for flipped kernels when the height or width is one, including in the reference implementation, but this does not happen too often and the total amount of work is still typically $1.2 \times 10^6 \cdot n$ FLOPs.

We briefly consider two techniques from numerical computing: the Fast Fourier Transform (FFT) as a route to faster convolutions, and methods for fast matrix multiplication. For the first, the idea fails immediately

*$T = 55$ was originally chosen to match the approximate dimensionality of the SRM features.[4]

†We count floating-point operations (FLOPs), although some can be done using integer arithmetic. Note that all modern processors, CPU or GPU, contain a fused multiply-add operation $x := x + y \times z$, so the convolution of n elements with a filter of size m requires approximately mn FLOPs.

because of the need to perform inverse transforms before computing histograms. The kernels are simply not big enough to make it worthwhile. For the second, suppose that each kernel is the same size, m pixels, and that there are k of them to perform. We can cast the entire computation as a single massive matrix multiplication KP where K is the $k \times m$ matrix with all the kernels as rows, and P is a $m \times n$ matrix which picks out the pixels corresponding to each kernel as its columns. Unfortunately, efficient matrix multiplication algorithms like Coppersmith's[8] only work well when all the dimensions k, m, n are reasonably large: in our case, the small m makes brute-force multiply the best practical method. The structure of P may offer some improvements, but we are not aware that this has been exploited in the literature.

We conclude that we must simply find the fastest way to perform well over 1 TFLOP (10^{12} FLOPs) of computation that is required to extract PSRM features from a 1Mpix image.

3. THE NVIDIA KEPLER ARCHITECTURE

The implementation is targeted at the third-generation NVIDIA GPGPU architecture *Kepler*; our hardware is the Tesla K20 card, designed for scientific computation and which can be bought for around $2800 today. The implementation is in CUDA, an extension of the C language which also compiles code for the GPU.[9] Some close-to-the-metal intrinsics are available.

For the purposes of this paper, the key characteristics of the architecture are as follows. Computations take place in *threads*, which execute as simultaneous identical instructions on 32 arithmetical units in parallel: a group of 32 such threads, which runs on one multiprocessor, is called a *warp*. We emphasise that the units in the warp must compute identical instructions in lock-step (although some instructions can execute conditionally), and effective programming avoids branches within warps. Warps are grouped into *blocks*, which can be distributed amongst (in this case) 78 multiprocessors, for a total of 2496 parallel cores. Blocks are interleaved in parallel. In theory the GK110 processor on the K20 card, clocked at 706Mhz, can schedule two single-precision operations per clock cycle per core and thus throughput 3.52TFLOP/s, but the limiting factor is usually memory bandwidth: getting the data into the registers fast enough to feed the compute units.

Efficient programming is strongly influenced by the different types of memory available to the processors. For our purposes we can consider three types: *global* memory is stored on the graphics card (DDR5), and has a latency of 200-400 clock cycles; *shared* memory is on-chip, has a latency of only a few cycles (depending on access patterns), is shared between all threads in the same block, but there is a maximum of 48KB available per block; *registers* are stored on-chip and have zero latency. The GPU attempts to hide global memory access latency by interleaving the computation of many blocks on each multiprocessor: this is sensible because each has 65536 registers that can be shared in banks between the computations, allowing context switch in negligible time. (For our purposes, the time required to copy memory from the host system to the global memory on the graphics card, and vice versa, is negligible.)

In implementation one first has to minimize the number of accesses to global memory, second to optimize the memory access pattern (of which more later), and finally to trade off higher parallelism against register/shared-memory exhaustion. Too many interleaved computations cause registers to spill to the L2 cache (which is as slow as global memory), too few expose the expensive memory-access latency. For more on writing CUDA code that runs efficiently, see for example Ref. 9.

4. IMPLEMENTING PSRM ON THE GPU

Some aspects of the PSRM features are unfavourable for parallel GPU processing. We first slightly adjust the features to remove some of the difficulties, before discussing how they are implemented.

4.1 Modifications to the features

The biggest drawback of the PSRM features is the random kernel size: it is difficult to optimize the inner loops when their size is not known at compile time, and also inconvenient to have threads process different-size filters. So, instead of applying convolutions of random size 1×1 to 8×8, we implement a fixed 4×4 size (which is a little below the average size of the original). In the conference paper originally proposing PSRM,[4] as opposed to

its refined journal version,[5] the projections were almost all contained within a 4×4 block anyway. If necessary one could include rows or columns of zeros to simulate smaller filters, but this seems unnecessary.

Second, we decided to use the same T random projections for each submodel. This means that only $16T$ random numbers (Gaussians scaled to unit length) are required. We also chose to cache them in shared memory, for faster access. We call the features with fixed-size kernels and fixed projections *GPU-PSRM* features.

Additionally, we consider reducing the parameter T, meaning fewer random projections per submodel. This is easily adjustable and practitioners can experiment with it, but the reduced time (roughly linear in T) can be significant, and the cost in terms of detection power seems negligible. One could also investigate using fewer submodels (also a potential linear time reduction), but we postpone this for future work.

These modifications certainly reduce the variety of the features, and could compromise their steganalysis power. In Sec. 5.2 we perform an experiment to check that the lost power is very small.

4.2 Optimizations

Our implementation uses the GPU only for the most intensive task: 'Step 2' of Sec. 2, computing convolutions and the resulting histograms. Filtering the input image, creating the residuals, and most of the summing of pre-features, is done using conventional CPU programming and, where possible, integer arithmetic.

Take a single residual, and suppose that it is $w_r \times h_r$ pixels in size. To exploit the thousands of arithmetical units available, it is broken up into tiles: each thread deals with one tile. It is convenient to divide the width of the residual amongst the 32 threads which make up one warp; vertically, the residual is divided into Θ intervals and the Θ warps make up one block. The Θ parameter may be varied to optimize the parallelism/memory tradeoff. Therefore each tile is $w_t = \lceil w_r/32 \rceil$ by $h_t = \lceil h_r/\Theta \rceil$ pixels in size, and a small amount of padding is added to the residual to make it up to a multiple of the tile size (at the final stage, before counting histograms, some conditional instructions are used to discount the padding). Each thread computes the output of a 4×4 convolution across a tile: it therefore needs to access $(w_t + 3) \times (h_t + 3)$ pixels of the residual. The breakdown of a residual into tiles, and the memory access of a thread, are shown in Fig. 2.

Each convolution consists of 16 multiply-add operations, but there is no need to write the result back to memory: once computed, it can be quantized and a count added to the relevant histogram bin (of which more in a moment). The fixed 4×4 kernel can be cached in shared memory, or, better kept in 16 registers. The slowest component is reading the 4×4 pixels from the residual, and here we can make a few optimizations. To get these optimizations to work, a certain amount of loop unrolling and code rewriting is necessary, but we leave this and other minor details out of this paper.

(i) Each pre-feature sums the histogram counts for $R * K$, $R * K^{\updownarrow}$, $R * K^{\leftrightarrow}$, and $R * K^{\updownarrow\leftrightarrow}$. Since these involve the same pixels, they can be computed together.

(ii) Some submodels only require the sum of these pre-features with the same kernel transposed (82 of the 168 residuals are used only in this way): these can also be computed together, saving memory access.

(iii) Each thread begins by loading the upper-left 4×4 pixels into 16 registers. Subsequently, it can scan down the tile by reading only four additional pixels and shuffling the others. At this point we only require 4 global memory reads for every 64 or 128 multiply-add operations.

(iv) Ideally we should also compute all T kernel convolutions at once, while we have the 16 residual pixels in registers. Unfortunately, this simply exchanges reading pixels for reading kernel elements, unless the latter can all be stored in registers or local memory, and it also makes more demands of register/local memory for counting simultaneous histograms. We parameterise this by computing Π convolutions by each thread.

Putting these together, each block consists of 32Θ threads which compute Π convolution-histograms for one residual. For the entire PSRM features we run $168\lceil T/\Pi \rceil$ blocks on the GPU.

There is also a considerable performance gain if we optimize the memory access patterns. When the 32 threads that make up a warp request consecutive words from global memory (and subject to some alignment constraints that do not concern us here), the entire block can be retrieved in a single access called a *coalesced*

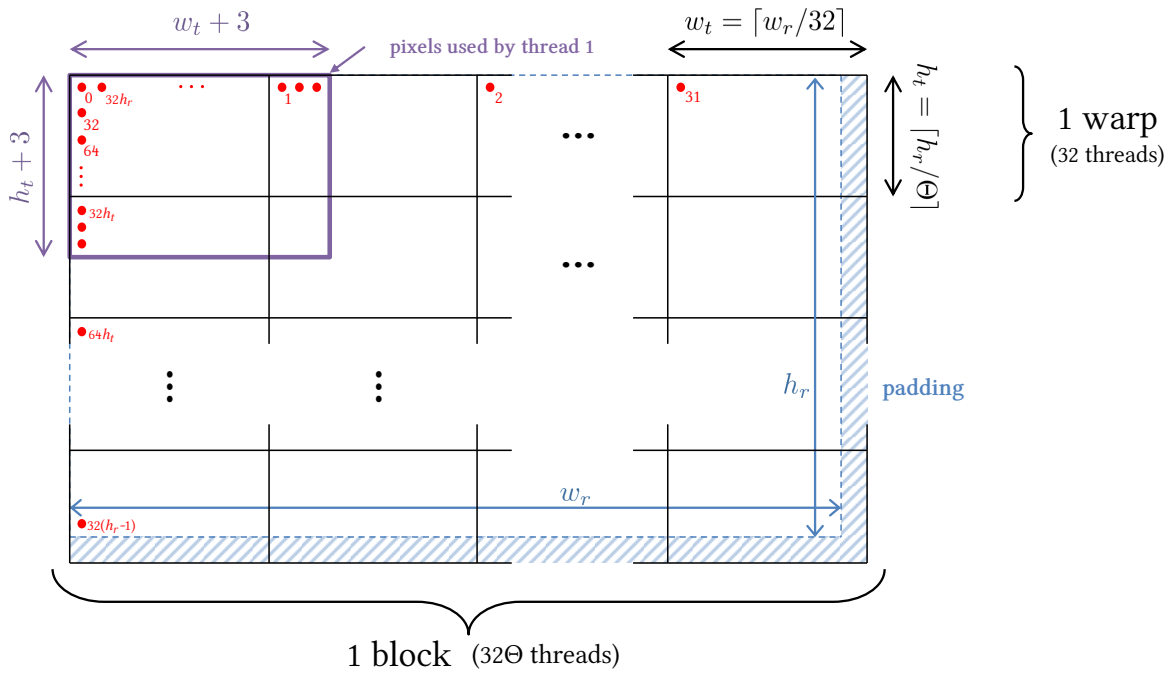

Figure 2. How each residual is divided into threads and warps, and (in red) the address offsets of the pixels which optimizes the memory access pattern. Pixels in the horizontally-overlapping regions are addressed by two locations: for example, the pixel at offset '1' (top-left of the second thread) is also at offset '$32h_r w_t$' (third from the top-right of the first thread).

transaction. If we store the residual in a conventional row-scan order, this does not happen, but we can ensure that all global accesses are coalesced if we re-order its pixels:

$$\text{memory offset } (i \cdot 32h_r + j \cdot 32 + k) \text{ stores residual at row } j, \text{ column } (k \cdot w_t + i).$$

Figure 2 shows the memory offsets of some of the pixels. Note that pixels in the overlapping parts of tiles may be stored twice over, once for each tile that accesses it: a minor increase in memory requirement.

Performing multiple convolutions is highly parallelizable; computing histograms is not. But for memory bandwidth reasons we need to do both within the GPU. It would be a disaster to write back all the convolution results to global memory, when all we care about is the central 6 histogram bins. Bearing in mind that histograms involve totalling over all the threads working on each residual, there are three sensible options for storing them:

(i) Use shared memory; each thread must maintain 6Π (or, for residuals where the transposed and untransposed kernels are counted separately, 12Π) histogram bins counting its tile, and at the end of the computation one thread adds them all together.

(ii) Use shared memory and have each thread increment global histogram bins. This is only possible because some atomic memory operations (which avoid concurrent update problems) are provided by CUDA.

(iii) Use 6Π (or 12Π) *registers* to keep the histogram counts in each thread, and atomic memory operations to total them at the end of the computation.

Option (iii) is the quickest, by quite a margin. The reason is that a) atomic memory operations are not very fast, and in any case shared memory is not as fast as registers, and b) there is simply not enough shared memory to run many such interleaved blocks on each multiprocessor, so the global memory access latency cannot be covered. However, option (iii) is also difficult to implement, because the compiler can only use registers if it known, at compile time, which operation increments which register. Figure 3 (left) cannot use registers, and we must use the apparently-inefficient code in Fig. 3 (right). Each bin increment compiles to just two instructions

```
% x is the convolution output for one pixel
bin=(int)floor(x);
histogram[bin]++;
```

```
% x is the convolution output for one pixel
bin=(int)floor(x);
if(bin==0) histogram[0]++;
if(bin==1) histogram[1]++;
  ⋮
```

Figure 3. Abstracted implementation of the histogram increment code: on the left, the compiler must store the histogram in global or shared memory. Although the code on the right appears less efficient, the histogram array can be stored in registers, and runs much faster.

and no branches: set a flag if equality, and conditionally increment a register if the flag is set. Here we only need 6 bins (-3 to $+3$).

In total, each thread requires at least 16Π registers to keep the kernel elements, a further 16 to store the currently-loaded pixels, 6Π or 12Π to store the histograms, plus about 20 others to keep track of location, addresses, flags, thread overhead, and temporary storage. Increasing Π increases the register requirement, and decreases the number of blocks that must be run concurrently, but decreases the need to re-read residual elements for different projections. Increasing Θ increases the number of concurrent threads per block and can lead to register exhaustion, but creates more blocks for interleaving. These parameters can be optimized at compile-time (we found $\Pi = 1$ and $\Theta \approx 4$ best for our hardware).

We also investigated two other optimizations, but discarded them because they did not increase performance: we tried caching three columns of each tile in shared memory, so that each pixel of a tile is read once only. This exhausted shared memory. We also tried generating the kernel elements directly each time they are used, thus replacing 16Π registers storing the elements with a few to store the state of a PRNG. But putting even highly efficient code for random Gaussians in the inner loop only makes it slower. If uniformly distributed kernel elements would suffice, this might be worth revisiting, but we did not investigate further.

5. EXPERIMENTAL RESULTS

Our first experiment was to validate that the PSRM-GPU implementation gives correct results, by comparing its output with that of the C++ reference implementation. By default, their output is completely different because they use different random number generators for the random kernels; naturally, PSRM features are only compatible when the same kernels are used. After fixing both implementations to identically-sized kernels with the same entries, we still observed some discrepancies between the resulting features.

This has two causes. The exact results of floating-point arithmetic depend on the implementation, and particularly the rounding mode, which may be different in CPUs and GPUs (and indeed across different models of CPU). Convolution results close to the quantization bin boundary can be flipped across it by different rounding during the multiply-and-add accumulation of the convolution. But such a change happens rarely, and causes only tiny differences in the resulting features. The second cause is a minor bug in the C++ implementation, where it is assumed that no convolution output is ever exactly an integer. The chances of this happening would be negligible, except that the content-suppression filters sometimes do such a good job that some blocks of residuals are all zero, and thus convolve to zero outputs. These differences are larger than those caused by rounding, but do not seem to affect the features' reliability for steganalysis.

By default the PSRM features are integers (because they sum various histogram counts), and their scale therefore depends on the size of the input image. This causes cover source mismatch when working with images of different sizes: to manage this, we recommend dividing the features by the number of pixels in the image.

5.1 Benchmarks

We now measure the speed of GPU-PSRM extraction, against existing implementations. All benchmarks were run on clusters in Oxford University's Advanced Research Computing (ARC) facility. The CPU cluster has 84 nodes each with 16 CPU cores, 2.0GHz Xeon SandyBridge processors; the GPU cluster has 8 nodes, each with 2 GK110 GPU processors at 706 MHz (only one of which was used) and 32 CPU cores, also 2.0GHz

Implementation	Time for one 1Mpix image
Reference PSRM, C++, single-thread	29588 s
Reference PSRM, MATLAB interpreted, multi-thread	2186 s CPU, 1100 s wallclock
Reference PSRM, MATLAB interpreted, single-thread	1554 s
Reference PSRM, MATLAB/MEX, single-thread	5753 s
Reference PSRM, MATLAB/MCC, multi-thread	2228 s CPU, 1109 s wallclock
Reference PSRM, MATLAB/MCC, single-thread	1524 s
PSRM, hand-optimized C, single-thread	929 s*
GPU-PSRM, optimized CUDA	2.6 s
GPU-PSRM (reduced), optimized CUDA	1.6 s

Table 1. Benchmarks of the various PSRM implementations. *Extrapolated from benchmark for one submodel.

Xeon SandyBridge processors. With parallelism limited to 16 cores, comparisons between the systems should be completely fair.

For the benchmarks, the C++ version was compiled using GCC with all available optimizations. A new hand-optimized C version, which exploits vectorized x64 SSE instructions, was compiled using Intel's compiler. The CUDA code was compiled using NVIDIA's 'nvcc' suite. The MATLAB versions were tested as interpreted code, with the PSRM C++ code compiled into a dynamically-linked function using the MEX compiler, and with the entire program compiled to an executable using MCC and Intel's compiler. We also switched MATLAB's multithreading capability on and off. The benchmarks in Tab. 1 show the average time to extract PSRM from 1Mpix images, including file I/O but excluding one-off startup time for the process.

The C++ reference implementation was written for readability, not speed, and it wastes a lot of time making one method call (which means building a stack frame) for every array access. MATLAB already uses highly-optimized implementations of convolutions, but still takes around 25 minutes for a single 1Mpix image. It can distribute some of its calculation amongst multiple cores, but for a problem of this size is only able to exploit 2-3 cores at a time and does so relatively inefficiently: the wallclock time falls to just below 20 minutes. It is possible that a differently-written implementation could improve on this parallelism. MATLAB compilation makes little difference. Optimized C code is difficult to write, and was only implemented for one submodel, but extrapolates to around 15 minutes of single-threaded code. The submodels are so trivially parallelizable that a fully-loaded 16-core machine might extract PSRM features in as little as one minute (but only if memory bandwidth is not limiting). This is still cumbersome for moderate- or large-scale experiments.

In the GPU-PSRM implementation we have successfully harnessed the parallel power of a GPU; the speed is over 400× faster than before on a wallclock basis (600× when compared with single-thread MATLAB), making PSRM features much more practically usable. Of the 2.6s wallclock time spent computing GPU-PSRM, the GPU kernels took about 1.7s and the CPU required 1.5s to compute the filters and re-order the residuals for coalesced access (some of this, obviously, was done in parallel).

The speedup is close to 1000× when the features are reduced to 30 projections per submodel. If we were to interleave the computations of multiple images, to maximize CPU/GPU parallelism, we should expect a throughput of better than one image per second (in the implementation of such parallelism it is easy to make mistakes, so we did not attempt it). Furthermore, we could double the throughput if we used both GPU cards on the cluster machine, trivial parallelism best exploited by running multiple processes.

5.2 Detection accuracy

Finally, we must check that the modifications to the GPU-PSRM features – kernel size fixed at 4×4, same random kernels for each submodel – does not negate their detection accuracy. We performed some steganalysis simulations, using the currently-leading ensemble classifier,[6] and measured the detection accuracy of PSRM and

Number of kernels per residual	Feature dimension	Mean testing error rate	Extraction time (256Kpix image)
GPU-PSRM implementation			
5	1170	16.48%	0.19 s
10	2340	15.71%	0.20 s
20	4680	14.88%	0.27 s
30	7020	14.78%	0.36 s
40	9360	14.75%	0.45 s
55	12870	14.34%	0.59 s
80	18720	14.29%	0.81 s
120	28560	14.19%	1.24 s
PSRM reference implementation (single-thread)			
55	12870	12.98%	491s

Table 2. Comparison of steganalysis accuracy (detecting HUGO embedding at 0.4 bits per pixel) and effect of T, the number of random projections per residual. For PSRM the multi-thread MATLAB implementation could have reduced wallclock time by about 30% for this size of image, but only by doubling the total CPU time.

GPU-PSRM on the 10000 images from BOSSBase v1.01.[10] The images are only 512×512 pixels, but have become a standard testbed in steganalysis literature.

The steganalysis method is HUGO, which PSRM features are particularly good at detecting,[4] with default parameters (which includes a payload of 0.4 bits per pixel). We did not test other payload sizes, embedding methods, or cover sources, because of the computational expense, but it is now easy for steganalysis researchers to compare GPU-PSRM with their own SRM/PSRM results. Repeating 10 times, 9000 of the images were used (in cover/stego pairs) for training an ensemble classifier. Its out of bag (OOB) error was optimized by varying the number of base learners and bagging dimensionality, and then its testing error was measured on the other 1000 cover/stego pairs. The overall average testing error is reported in Tab. 2.

We see that the reduced diversity of GPU-PSRM features *does* have a cost, but only around 1% additional error rate in this case. That difference is statistically significant over a sample of 20000 images, but not practically very significant: a practitioner would very likely trade 1% accuracy for a 10× speedup, let alone 400–600×. Moreover, there is negligible additional cost if the parameter T is reduced to around 20–30 instead of the default 55. This carries a dual advantage: quicker extraction of features, and the reduced dimensionality implies quicker training and application of classifiers that use the features (and probably smaller training set sizes as well).

We conclude with a fact that highlights the cost of using slow-to-extract features. The above experiment, which involved extracting PSRM features from 10000 cover and 10000 stego images each only 256Kpix, required 2732 core hours on the Oxford University ARC facility. At internal charging rates, this single experiment would cost £136 ($223); equivalent time on Amazon's EC2 would cost approximately twice as much at today's prices. Optimizing parameters and testing new embedding methods could become very expensive research. For comparison, the same experiment with GPU-PSRM took around 3.3 hours on one GPU, the capital cost of which could quickly be recouped.

6. CONCLUSION AND FUTURE DIRECTIONS

The PSRM features are the most computationally demanding yet created, but the parallel power of the GPU implementation renders them usable in experiments with thousands or millions of images. Moreover, slightly modifying the features to make them more optimization-friendly does not appear to have reduced their power much. The GPU-PSRM code is available from the author's website[11] and can be compiled for other NVIDIA GPUs, although it was designed for the *Kepler* architecture and the K20 card in particular. The Θ and Π

parameters can be varied, which may allow the code to work effectively on lower-spec graphics cards. Register exhaustion is likely to be the limiting factor.

We expect that it would be possible to optimize the implementation a little further, perhaps by generating the residuals on the GPU rather than the CPU, and by optimizing the pipeline for parallel images, but perhaps we should reflect on whether it is reasonable to perform such an enormous amount of computation at all. Given that PSRM is only marginally more powerful than its predecessors, it may be more valuable to settle for optimized implementations of SRM, JRM, and similar steganalysis features. Indeed, if steganalysis were to be used for real-time network scanning, the trade-off between speed and accuracy would have to be adjusted considerably, and much simpler features implemented. There has been very little research in this direction. As well as fast feature extraction it would be important to apply fast classifiers, an approach studied in Ref. 12.

Fast implementation need not necessarily involve a GPU: the latest generation CPU architecture (*Haswell*) approaches the floating-point capability of GPUs, as it can manage 16 multiply-adds per clock cycle per core. In theory a multi-core machine could compute almost 1 TFLOP/s with better memory caching than current GPU cards.

ACKNOWLEDGMENTS

Many thanks to Jan Kodovský and Vojtěch Holub, for sharing their reference implementations of JRM, SRM, and PSRM features. See `http://dde.binghamton.edu/download/feature_extractors/`. The author would like to acknowledge the use of the University of Oxford Advanced Research Computing (ARC) facility in carrying out the experiments. The work on this paper was supported by European Office of Aerospace Research and Development under the research grant number FA8655-13-1-3020. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation there on. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of EOARD or the U.S. Government.

REFERENCES

[1] Pevný, T. and Fridrich, J., "Merging Markov and DCT features for multi-class JPEG steganalysis," in [*Security, Steganography, and Watermarking of Multimedia Contents IX*], *Proc. SPIE* **6505**, 03–14, SPIE (2007).

[2] Fridrich, J. and Kodovský, J., "Rich models for steganalysis of digital images," *IEEE Transactions on Information Forensics and Security* **7**(3), 868–882 (2012).

[3] Kodovský, J. and Fridrich, J., "Steganalysis of JPEG images using rich models," in [*Media Watermarking, Security, and Forensics 2012*], *Proc. SPIE* **8303**, 0A01–0A13, SPIE (2012).

[4] Holub, V., Fridrich, J., and Denemark, T., "Random projections of residuals as an alternative to co-occurrences in steganalysis," in [*Media Watermarking, Security, and Forensics 2013*], *Proc. SPIE* **8665**, 0L01–0L11, SPIE (2013).

[5] Holub, V. and Fridrich, J., "Random projections of residuals for digital image steganalysis," *IEEE Transactions on Information Forensics and Security* **8**(12), 1996–2006 (2013).

[6] Kodovský, J., Fridrich, J., and Holub, V., "Ensemble classifiers for steganalysis of digital media," *IEEE Transactions on Information Forensics and Security* **7**(2), 432–444 (2012).

[7] Lubenko, I. and Ker, A. D., "Going from small to large data in steganalysis," in [*Media Watermarking, Security, and Forensics 2012*], *Proc. SPIE* **8303**, 0M01–0M10, SPIE (2012).

[8] Coppersmith, D. and Winograd, S., "Matrix multiplication via arithmetic progressions," *Journal of Symbolic Computation* **9**(3), 251–280 (1990).

[9] Wilt, N., [*The CUDA Handbook*], Addison Wesley (2013).

[10] Bas, P., Filler, T., and Pevný, T., "'Break Our Steganographic System': The ins and outs of organizing BOSS," in [*Proc. 13th Information Hiding Workshop*], *LNCS* **6958**, 59–70, Springer (2011).

[11] Ker, A. D., "GPU-PSRM implementation," (2014). Available from `http://www.cs.ox.ac.uk/andrew.ker/gpu-psrm/`.

[12] Lubenko, I., *Towards Robust Steganalysis: Binary Classifiers and Large, Heterogeneous Data*, PhD thesis, University of Oxford (submitted) (2014).