Papers I have read

宋明辉

May 16, 2018

Contents

List of Figures

Usage Instructions:

- This book include all papers I have read from 2017.05.28.
- The magenta represent the online link.
- The red represents the links in this book including reference, figure, table and others.
- The purple represents the emphasize.

Chapter 1

Image processing based on CUDA

This chapter include the Image processing acceleration based on CUDA.

1.1 Novel multi-scale retinex with color restoration on graphics processing unit

1.1.1 Abstract

In this paper, a parallel application of the MSRCR+AL algorithm on a GPU is presented. For the various configurations in our test, the GPU-accelerated MSRCR+AL shows a scalable speedup as the resolution of an image increases. The up to $45\times$ speed up (1024×1024) over the single-threaded CPU counterpart shows a promissing direction of using the GPU-based MSRCR+AL in large scale, time-critical applications. We also achieved 17 frames per second in video processing (1280×720).

1.1.2 Content

In our implementation, the CUFFT provides a simple interface for computing FFTs. After the plans of both forward and inverse FFTs are created according to the CUFFT requirements, the image data and the Gaussian filters can be parallel transformed to frequency domain. The multiplication between image data and Gaussian filters in frequency domain is finished by ModulateandNormal- ize() function, which is also provided by the CUFFT library.

The atomic function atomicAdd() provided by CUDA is used in the kernel histogram function to guar- antee to be performed without interference from other threads.

1.1.3 Parallel optimization strage

size of thread block and grid

For example, the maximum number of threads on the lower capability version of CUDA is 512, but newer CUDA-enabled GPUs with 2.x computecapability can reach 1,024. But, each streaming multi- processer (SM) can only execute 1,536 threads simul-

taneously. Therefore, we set the number of threads per block at 192, which means each SM can fully execute eight blocks to maximize resources.

Memory access optimization

A thread needs 400–600 clock cycles to access the global memory, but only needs about 4 clock cycles to access fast memory units such as register and shared memory due to lower access latency. Therefore, taking full advantage of the multi-level GPU memory storage components can obtain quick data access to improve the execution performance effectively.

Loop unrolling

After loop unrolling, the code only needs to run one time to write the result. The number of write processes decreases 66.7% compared with the serial code. Also, it only needs one time to read the result, compared with three times had we used in the serial code. The number of read processes decreases 66.7% as well. Furthermore, this loop unrolling strategy can be applied to sum different scale results together for the Multi-scale Retinex. More importantly, in the reduction algorithm for the summation process, this strategy is used to greatly increase the calculation speed and reduce the instruction overhead.

1.1.4 Conclusion

see the Abstract subsection.

1.2 Image Convolution

This section includes all articles I have read relating to Image Convolution using CUDA. Image convolution is usually used in image filtering, like gaussian filter.

1.2.1 Naïve Implementation

From the idea of convolutio filter itself, the most naive approach is to use global memory to send data to device and each thread accesses this to compute convolution kernel. Our convolution kernel size is radius 8 (total 17×17 multiplicaiton for single pixel value). In image border area, reference value will be set to 0 during computation. This naive approach includes many of conditional statements and this causes very slow execution. The code is as shown below:

Listing 1.1: Image Convolution based on global memory

```
__global__ void gaussfilterGlo_kernel(float *d_imgOut, float *
    d_imgIn, int wid, int hei,
```

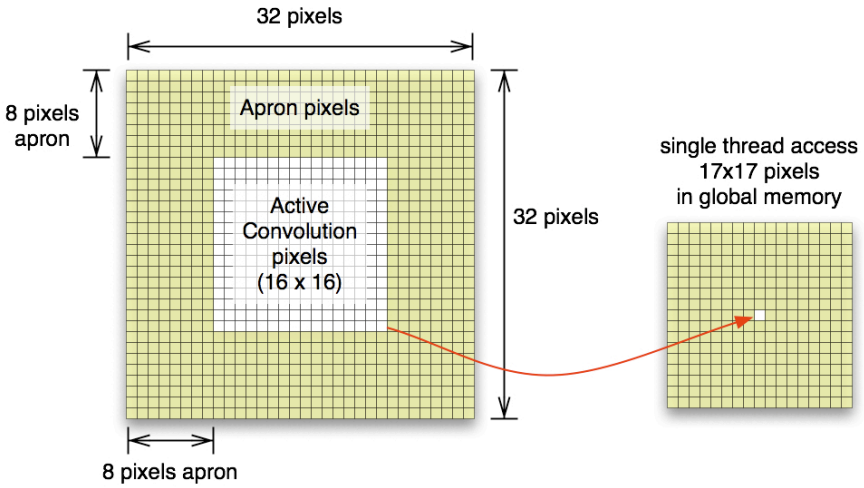

Fig. 1.1. Image Convolution based on Global Memory

```
{
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    int idy = threadIdx.y + blockDim.y * blockIdx.y;

    if(idx > wid || idy > hei)
        return ;

    int filterR = (filterW - 1) / 2;

    float val = 0.f;

    for(int fr = -filterR; fr<= filterR; ++fr)            // row
        for(int fc = -filterR; fc <= filterR; ++fc)       // col
        {
            int ir = idy + fr;
            int ic = idx + fc;

            if((ic >= 0) && (ic <= wid - 1) && (ir >= 0) && (ir <=
                hei - 1))
                val += d_imgIn[INDX(ir, ic, wid)] * d_filter[INDX(fr
                    +filterR, fc+filterR, filterW)];
        }
    d_imgOut[INDX(idy, idx, wid)] = val;
```

```
}
```

For $396 \star 396$ input image, the time is 1.6ms. When the input filter is stored in constant memory or specified by '__restrict__', the time is 1.7 or 1.8 ms.

1.2.2 Naïve Shared Memory Implementation

The simplest approach to implement convolution in CUDA is to load a block of the image into a shared memory array, do a point-wise multiplication of a filter-size portion of the block, and then write this sum into the output image in device memory. Each thread block processes one block in the image. Each thread generates a single output pixel.

The algorithm itself is somewhat complex. For any reasonable filter kernel size, the pixels at the edge of the shared memory array will depend on pixels not in shared memory. Around the image block within a thread block, there is an *apron* of pixels of the width of the kernel radius that is required in order to filter the image block. Thus, each thread block must load into shared memory the pixels to be filtered and the apron pixels.

Note: The apron of one block overlaps with adjacent blocks. The aprons of the blocks on the edges of the image extend outside the image – these pixels can either be clamped to the color of pixels at the image edge, or they can be set to zero.

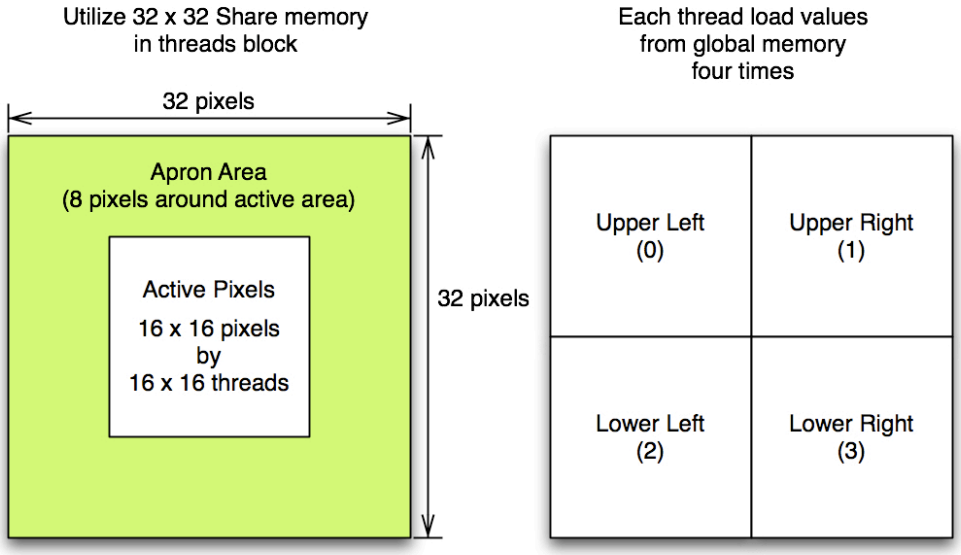

Fig. 1.2. Image Convolution based on shared memory

The first attempt was to keep active thread size as same as previous and increase block size for apron pixels. This did not work since convolution kernel radius is 8 and it make block size to 32×32 (1024). This is bigger than G80 hardware limit (512 threads max per block).

Therefore, I changes scheme as all threads are active and each thread loads four pixels and keep the block size 16×16. Shared Memory size used is 32×32 (this includes all necessary apron pixel values for 16×16 active pixels). Below shows quite a bit of performance improve. This is almost $\times 2.8$ speed up over naive approach (in 2048 resolution).

Listing 1.2: Image colvotion based on Shared memory and Apron

```
__global__ void convolutionGPU(
..............................float *d_Result,
..............................float *d_Data,
..............................int dataW,
..............................int dataH
..............................)
{
....// Data cache: threadIdx.x , threadIdx.y
....__shared__ float data[TILE_W + KERNEL_RADIUS * 2][TILE_W +
    KERNEL_RADIUS * 2];

....// global mem address of this thread
....const int gLoc = threadIdx.x +
.......................IMUL(blockIdx.x, blockDim.x) +
.......................IMUL(threadIdx.y, dataW) +
.......................IMUL(blockIdx.y, blockDim.y) * dataW;

....// load cache (32x32 shared memory, 16x16 threads blocks)
....// each threads loads four values from global memory into shared
     mem
....// if in image area, get value in global mem, else 0
....int x, y;   // image based coordinate

....// original image based coordinate
....const int x0 = threadIdx.x + IMUL(blockIdx.x, blockDim.x);
....const int y0 = threadIdx.y + IMUL(blockIdx.y, blockDim.y);

....// case1: upper left
....x = x0 - KERNEL_RADIUS;
....y = y0 - KERNEL_RADIUS;
....if ( x < 0 || y < 0 )
........data[threadIdx.x][threadIdx.y] = 0;
....else
........data[threadIdx.x][threadIdx.y] = d_Data[ gLoc -
    KERNEL_RADIUS - IMUL(dataW, KERNEL_RADIUS)];

....// case2: upper right
....x = x0 + KERNEL_RADIUS;
....y = y0 - KERNEL_RADIUS;
....if ( x > dataW-1 || y < 0 )
........data[threadIdx.x + blockDim.x][threadIdx.y] = 0;
....else
........data[threadIdx.x + blockDim.x][threadIdx.y] = d_Data[gLoc +
    KERNEL_RADIUS - IMUL(dataW, KERNEL_RADIUS)];

....// case3: lower left
....x = x0 - KERNEL_RADIUS;
....y = y0 + KERNEL_RADIUS;
....if (x < 0 || y > dataH-1)
........data[threadIdx.x][threadIdx.y + blockDim.y] = 0;
....else
........data[threadIdx.x][threadIdx.y + blockDim.y] = d_Data[gLoc -
    KERNEL_RADIUS + IMUL(dataW, KERNEL_RADIUS)];
```

```
....// case4: lower right
....x = x0 + KERNEL_RADIUS;
....y = y0 + KERNEL_RADIUS;
....if ( x > dataW-1 || y > dataH-1)
........data[threadIdx.x + blockDim.x][threadIdx.y + blockDim.y] =
    0;
....else
........data[threadIdx.x + blockDim.x][threadIdx.y + blockDim.y] =
    d_Data[gLoc + KERNEL_RADIUS + IMUL(dataW, KERNEL_RADIUS)];

....__syncthreads();

....// convolution
....float sum = 0;
....x = KERNEL_RADIUS + threadIdx.x;
....y = KERNEL_RADIUS + threadIdx.y;
....for (int i = -KERNEL_RADIUS; i <= KERNEL_RADIUS; i++)
........for (int j = -KERNEL_RADIUS; j <= KERNEL_RADIUS; j++)
............sum += data[x + i][y + j] * d_Kernel[KERNEL_RADIUS + j]
    * d_Kernel[KERNEL_RADIUS + i];

....d_Result[gLoc] = sum;
}
```

Note : the value "gLoc - KERNEL_RADIUS - IMUL(dataW, KERNEL_RADIUS)" is the shift address of the image data on the upper left corner. 在本方法中，主要是索引的问题，所以又分为 Share Memory 的索引以及图像数据的索引。在具体实现过程中，选择是固定 thread Block 的大小，同时在 share memory 中添加边界。所以将图像数据拷贝到 Share Memory 中时，分了四次，分别对应：左上角，右上角，左下角、右下角。也就是图1.2中的对应关系，其处理过程就是将小的图像块映射到大的 Share memory 中，从这个方面进行理解。

1.2.3 Separable Gaussian Filtering

Separable Convolution

A tow-dimensional filter s is said to be separable if it can be written as the convolution of tow one-dimesional filters ***v*** and ***h***:

$$s = v * h$$

"How to determine if a matrix is an outer product of two vectors?"
"Go look at the **rank** function. ". Of course. If a matrix is an outer product of two vectors, its rank is 1.

So the test is this: The rank of A is the number of nonzero singular values of A, with some numerical tolerance based on eps and the size of A.

So how can we determine the outer product vectors? The answer is to go back to the svd function. Here's a snippet from the doc:

$[U, S, V] = svd(X)$ produces a diagonal matrix S of the same dimension as X, with nonnegative diagonal elements in decreasing order, and unitary matrices ***U*** and ***V*** so that $X = U * S * V'$

A rank 1 matrix has only one nonzero singular value, so $X = U * S * V'$ becomes $U(:,1) * S(1,1) * V(:,1)$. This is basically the outer product we were seeking. Therefore, we want the first columns of ***U*** and ***V***. (We have to remember also to use the nonzero singular value as a scale factor.)

Seperate Gaussian filter

First chose, somewhat artitrarily to split the scale factor, S(1, 1), "equally" between ***v*** and ***h***. Except for normal floating-point roundoff differences, gaussian and v*h are equal. Just show as following :

$$[U, S, V] = svd(X)$$
$$v = U(:,1) * sqrt(S(1,1))$$
$$h = V(:,1)`\times sqrt(S(1,1))$$
$$GassianFilter = v * h$$

More details can be found at :Separable Convolution.

1.2.4 Optimizing for memory coalescence

Base read/write addresses of the warps of 32 threads also must meet half-warp alignment requirement in order to be coalesced. If four-byte values are read, then the base address for the warp must be 64-byte aligned, and threads within the warp must read sequential 4-byte addresses. If the dataset with apron does not align in this way, then we must fix it so that it does.

The approach used in the row filter is to have additional threads on the leading edge of the processing tile, in order to make threadIdx.x == 0 always reading properly aligned address and thus to meet global memory alignment constraints for all warps. This may seem like a waste of threads, but it is of little importance when the data block, processed by a single thread block is large enough, which decreases the ratio of apron pixels to output pixels.

Each image convolution pass in both row and column pass is separated into two sub stages within corresponding CUDA kernels. The first stage loads the data from global memory into shared memory, and the second stage performs the filtering and writes the results back to global memory. We mustn’ t forget about the cases when row or column processing tile becomes clamped by image borders, and initialize clamped shared memory array indices with correct values. Indices not lying within input image borders are usually initialized either with zeroes or with values, corresponding to clamped image coordinates. In this sample we opt for the former.

In between the two stages there is a **__syncthreads( )** call to ensure that all threads have written to shared memory before any processing begins. This is necessary because threads are dependent on data loaded by other threads.

For both the loading and processing stages each active thread loads/outputs one pixel. In the computation stage each thread loops over a width of twice the filter radius plus 1, multiplying each pixel by the corresponding filter coefficient stored in constant memory. Each thread in a half-warp accesses the same constant address and hence there is no penalty due to constant memory bank conflicts. Also, consecutive threads always access consecutive shared memory addresses so no shared memory bank conflicts occur as well.

The column filter pass operates much like the row filter pass. The major difference is that thread IDs increase across the filter region rather than along it. As in the row filter pass, threads in a single half-warp always access different shared memory banks, but the calculation of the next/previous addresses involves increment/decrement by COLUMN_TILE_W, rather than simply 1. In the column filter pass we do not have inactive "coalescing alignment" threads during the load stage, because we assume that the tile width is a multiple of the coalesced read size. In order to decrease the ratio of apron to output pixels we want image tile to be as tall as possible, so to have reasonable shared memory utilization we shoot for as thin image tiles as possible: 16 columns.

1.3 CUDA C Best Practice Guide

1.3.1 Performance Metrics

Timing

- Using CPU Timers
  Should call *cudaDeviceSynchronize()* immediately before starting and stopping the CPU timer.

- Using CUDA GPU Timers
  The device will record a timestamp for the event when it reaches that event in the stream. This value is expressed in milliseconds and has a resolution of approximately half a microsecond.

Bandwidth

Bandwidth - the rate at which data can be transferred - is one of the most important gating factors for performance.

1.4 Npp Library Image Filters

1.4.1 Image Data

Line Step

All image data passed to NPPI primitives requires a line step to be provided. It is important to keep in mind that this line step is always specified in terms of bytes, not pixels.

1.5 PTX ISA 6.0

1.5.1 PTX Machine Model

The *Multiprocessor* maps each thread to one *scalar processor* core, and each scalar thread executes independently with its own instruction address and register state.

Individual threads composing a SIMT warp start together at the same program address but are otherwise free to branch and execute independently. A warp executes one common instruction at a time, so full efficiency is realized when all threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path.

Each multiprocessor has **on-chip memory** of the four following types :

- Local 32-bit registers per processor;
- Shared memory (parallel data cache);
- Read-only *constant cache* that is shared by all scalar processor cores;
- Read-only *texture cache*

The local and global memory spaces are read-write regions of **device memory** and are not cached.

If there are not enough registers or shared memory available per multiprocessor to process at least one block, the kernel will fail to launch.

Syntax

PTX programs are a collection of text source modules(files). PTX source modules have an assembly-language style syntax with instruction operation codes and operands. Pseudo-operations specify symbol and addressing management. The *ptxas* optimizing backend compiler optimizes and assembles PTX source modules to produce corresponding binary object files.

Source Format

PTX is case sensitive and uses lowercase for keywords.

Each PTX module must begin with a **.version** directive specifying the PTX language version, followed by a **.target** directive specifying the target architecture assumed.

Statements

A PTX statement is either a **directive** or an **instruction**. Statements begin with an optional label and end with a semicolon.

- Directive Statements
  Directive keywords begin with a dot, so no conflict is possible with user-defined identifiers. 如图1.3所示。
- Instruction Statements
  Instructions are formed from an instruction opcode followed by a **comma-separated** list of zero or more operands, and terminated with a semicolon.

Table 1 PTX Directives

.address_size	.file	.minnctapersm	.target
.align	.func	.param	.tex
.branchtargets	.global	.pragma	.version
.callprototype	.loc	.reg	.visible
.calltargets	.local	.reqntid	.weak
.const	.maxnctapersm	.section	
.entry	.maxnreg	.shared	
.extern	.maxntid	.sreg	

Fig. 1.3. PTX Directives

Operands may be register variables, constant expressions, address expressions, or label names. The guard predicate follows the optional label and precedes the opcode, and is written as **@p** , where p is a predicate register. The guard predicate may be optionally negated, written as **@!p**.

The destination operand is first, followed by source operands. 如图1.4所示。

Table 2 Reserved Instruction Keywords

abs	div	or	sin	vavrg2, vavrg4
add	ex2	pmevent	slct	vmad
addc	exit	popc	sqrt	vmax
and	fma	prefetch	st	vmax2, vmax4
atom	isspacep	prefetchu	sub	vmin
bar	ld	prmt	subc	vmin2, vmin4
bfe	ldu	rcp	suld	vote
bfi	lg2	red	suq	vset
bfind	mad	rem	sured	vset2, vset4
bra	mad24	ret	sust	vshl
brev	madc	rsqrt	testp	vshr
brkpt	max	sad	tex	vsub
call	membar	selp	tld4	vsub2, vsub4
clz	min	set	trap	xor
cnot	mov	setp	txq	
copysign	mul	shf	vabsdiff	
cos	mul 24	shfl	vabsdiff2, vabsdiff4	
cvt	neg	shl	vadd	
cvta	not	shr	vadd2, vadd4	

Fig. 1.4. Reserved Instruction Keywords

Chapter 2

FPGAs

Every section is arranged like follows:

- Abstract
  The abstract part of paper.
- Content
  The main idea of paper.
- Results
  The experiment implementation.
- Conclusion
  The conclusion part of paper.

2.1 Performance Comparison of FPGA, GPU and CPU in Image Processing 2009

2.1.1 Abstract

Many applications in image processing have high inherent parallelism. FPGAs have shown very high performance in spite of their low operational frequency by fully extracting the parallelism. In recent micro processors, it also becomes possible to utilize the parallelism using multi-cores which support improved SIMD instructions, though programmers have to use them explicitly to achieve high performance. Re- cent GPUs support a large number of cores, and have a po- tential for high performance in many applications. **However, the cores are grouped, and data transfer between the groups is very limited.** Programming tools for FPGA, SIMD in- structions on CPU and a large number of cores on GPU have been developed, but it is still difficult to achieve high perfor- mance on these platforms. In this paper, we compare the per- formance of FPGA, GPU and CPU using three applications in image processing; two-dimensional filters, stereo-vision and k-means clustering, and make it clear which platform is faster under which conditions.

Fig. 2.1. 传统提升小波计算过程

2.1.2 Content

Compared three applications: two-dimension filters, stereo-vision, k-means clustering.

The high performance of FPGA comes from its flexibility which makes it possible to realize the fully optimized circuit for each application, and a large number of on-chip memory banks which supports the high parallelism.**FPGA can achieve extremely high performance in many applications in spite of its low op- erational frequency**.

GPU cores are grouped, the data transfer between groups is very slow.

GPU Analysis

It consists of 10 thread processor clus- ters. A thread processor cluster has three streaming mul- tiprocessors, eight texture filtering units, and one level-1 cache memory. Each streaming multiprocessor has one in- struction unit, eight stream processors (SPs) and one local memory (16KB). Thus, GTX280 has 240 SPs in total. Eight SPs in a stream-

ing multiprocessor are connected to one in- struction unit. This means that the eight SPs execute the same instruction stream on different data.

Two-Dimensional Filters

The computational complexity of filters is $O(w \times w)$, and w is radius of filter.

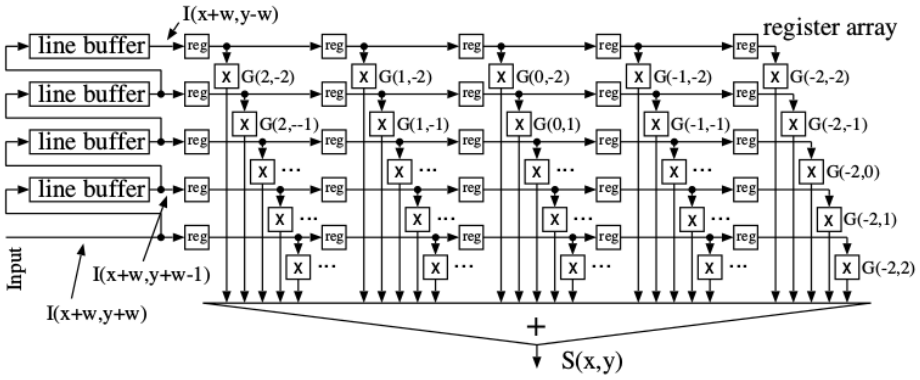

Fig. 2.2. Circuits for non-separable filters

Fig2.2 shows the filter is 5×5 case.

Stereo Vision

This application is to get the distance to the location obtained from the two camera's disparity.The sum of absolute difference (SAD) is widely used to compared the windows because of its simplicity. More details can be found in paper[1]. **K-means Clustering**

More details can be found in paper [1].

2.1.3 Results

- Xilinx XC4VLX160.
- GeForce 280GTX, 1 GB DDR3, CUDA version 2.1.
- Intel Core2 Extreme QX6859.

The time to download images from main memory is not included. CPU has four cores, FPGA is fixed to 100MHz. Fig is the performance of two-dimension filters.

GPU is the fastest for all tested filter size.In this problem, filters can be applied to each pixel in the image in- dependently without using shared variables. So, GPU can show its best performance.

In the later two applications, the performance FPGA is much better than GPU.

2.1.4 Conclusion

We have compared the performance of GPU with FPGA and CPU (quad-cores) using three simple problems in im- age processing. GPU has a potential for achieving almost the same performance with FPGA. The number of cores in GTX280 is 240.Considering the trade-offs between the operational frequency of GPU (more than 10 times faster), and

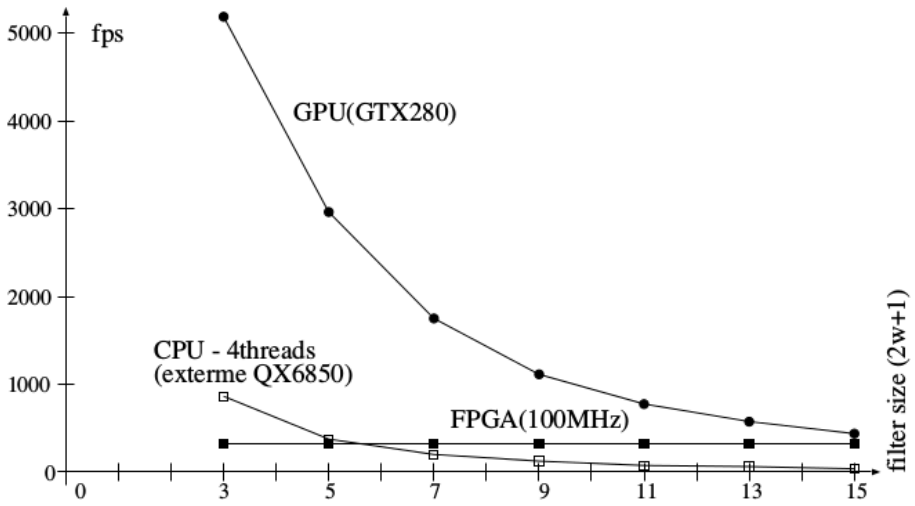

Fig. 2.3. Performance of two-dimensional filters

the fine-grained parallelism in FPGA, this seems to be a natural consequence. However, GPU can show its potential only for naive computation methods, in which all pixels can be processed independently. For more sophisticated algorithms which use shared arrays, GPU can not execute those algorithms because of its very small local memory, or can not show good performance because of the memory access limitation caused by its memory architecture. GPU is slower than CPU in those algorithms (it may be possible to realize much better performance if we can find algorithms which can get around the limitations, but we could not find them). The performance of CPU is 1/12 - 1/7 of FPGA, which means that CPU with quad-cores can executes about 1/10 operations of FPGA in a unit time (the same algorithms are executed on CPU and FPGA). The performance of FPGA is limited by the size of FPGA and the memory bandwidth. With a latest FPGA board with DDR-II DRAM and a larger FPGA, it possible to double the performance by processing twice the number of pixels in parallel.

We have the following issues which have to be consid- ered. We have compared the performance using only three problems. The performances of the programs on GPU and CPU are not fully tuned up. In the comparison, power con- sumption and costs are not considered.

2.2 Fast FPGA Prototyping for real-time image processing with very high-level synthesis 2017

2.2.1 Abstract

Programming in high abstraction level can facilitate the development of digital signal processing systems. In the recent 20 yeahs, HLS has made significantly progress. However, due to the high complexity and computational intensity, image processing algorithms usually necessitate a higher abstraction environment than C-synthesis, and the current HLS tools do not have the ability of this kind. This paper presents a conception of

very high-level synthesis method which allows fast prototyping and verifying the FPGA-based image processing designs in the MATLAB environment. We build a heterogeneous development flow by using currently available tool kits for verifying the proposed approach and evaluated it within two real-life applications. Experiment results demonstrate that it can effectively reduce the complexity of the development by automatically synthesizing the algorithm behavior from the user level into the low register transfer level and give play to the advantages of FPGA related to the other devices.

2.2.2 Content

Advanced Digital Sciences Center(ADSC) of the University of Illinois reported that FPGA can achieve a speedup to 2-2.5x and save 84-92% of the energy consumption comparing to Graphics Processing Units(GPUs). ADSC indicates also that a manual FPGA design may consume 6-18 months and even years for a full custom hardware, while the GPUs(CUDA) based designed only 1-2 weeks.

Fig2.4 show the Gasjki-Kuhn's Y-chart comparing the conventional RTL with the HLS-based design flows.
The challenges of MATLAB-to-RTL synthesis include:

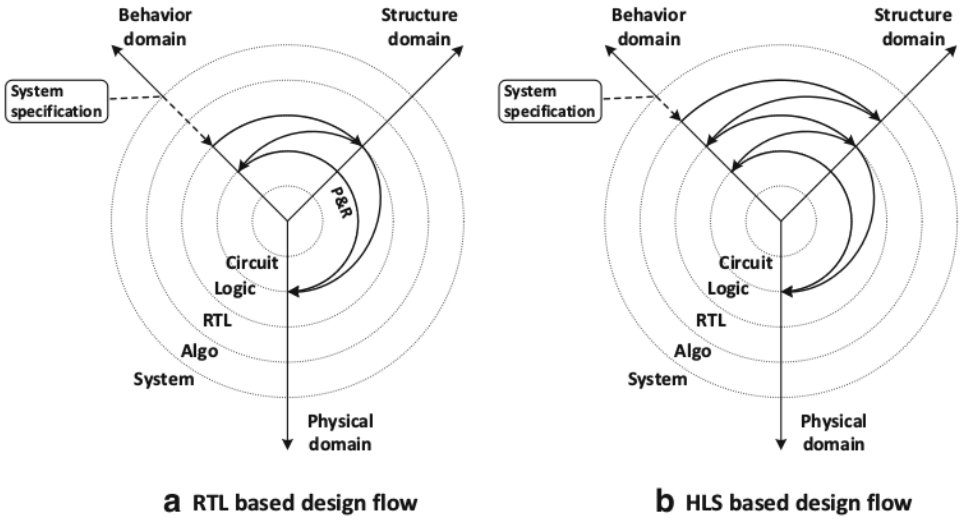

Fig. 2.4. Comparison of RTL- and HLS-based design flows by using Gasjki-Kuhn's Y-chart: full lines indicate the automated cycles, while dotted lines the manual cycles.

- Operators in MATLAB perform different operations depending on the type of the operands, whereas the functions of the operators in RTL are fixed.
- MATLAB includes very simple and powerful vector operations such as the concatenation ``[ ]" and column operators ``$x(:)$" or ``end" construct, which can be quite hard to map to RTL.

- MATLAB supports ``polymorphism'' whereas RTL does not. More precisely, functions in MATLAB are genetic and can process different types of input parameters. In the behaviors of RTL, each parameter has only a single given type, which cannot change.
- MATLAB supports dynamic loop bounds or vector size, whereas RTL requires users to initialize explicitly them and cannot do any changes during the synthesis.
- The variables in MATLAB can be reused for different contents (different types), whereas RTL does not, as each variable has one unique type.

Two complex image processing applications: Kubelka-Munk genetic algorithm(KMGA) for the multispectral image based skin lesion assessments & level set method(LSM)-based algorithm for very high-resolution(VHR) satellite image segmentation..

Chapter 3

Image Fusion

3.1 Guided Image Filter 2013

3.1.1 Abstract

In this paper, we propose a novel explicit image filter called guided filter. Derived from a local linear model, the guided filter computes the filtering output by considering the content of a guidance image, which can be the input image itself or another different image. The guided filter can be used as an edge-preserving smoothing operator like the popular bilateral filter, but it has better behaviors near edges. The guided filter is also a more generic concept beyond smoothing: It can transfer the structures of the guidance image to the filtering output, enabling new filtering applications like dehazing and guided feathering. Moreover, the guided filter naturally has a fast and nonapproximate linear time algorithm, regardless of the kernel size and the intensity range. Currently, it is one of the fastest edge-preserving filters. Experiments show that the guided filter is both effective and efficient in a great variety of computer vision and computer graphics applications, including edge-aware smoothing, detail enhancement, HDR compression, image matting/feathering, dehazing, joint upsampling, etc.

3.1.2 Content

A general linear translation-variant filtering process, which involves a guidance image *I*, an filtering input image *p* and an output image *q*. Both I and p are given beforehand according to the application, and they can be **identical**! The filtering output at a pixel *i* is expressed as a weight average:

$$q_i = \sum_j W_{ij}(I) p_j \tag{3.1}$$

where *i, j* are pixel indexes. The filter kernel W_{ij} is a function of the guidance image I and independent of p.

3.1.3 Conclusion

3.2 Multiscale Image Fusion Through Guided Filtering

3.2.1 Abstract

We introduce a multiscale image fusion scheme based on GUided Filtering. Guided filtering can effectively reduce noise while preserving details boundaries. While restoring larger scale edges. The proposed multi-scale fusion scheme achieves optimal spatial consistency by using guided filtering both at the decomposing and at the recombination stage of the multiscale fusion process. First, size-selective iterative guided filtering is applied to decompose the source images into base and detail layers at multiple levels of resolution. Next, at each resolution level a binary weighting map is obtained as the pixelwise maximum of corresponding source saliency maps. Guided filtering of the binary weighting maps with their corresponding source images as guidance images serves to reduce noise and to restore spatial consistency. The final fused image is obtained as the weighted recombination of the individual detail layers and the mean of the lowest resolution base layers. Application to multiband visual (intensified) and thermal infrared imagery demonstrates that the proposed method obtains state-of-the-art performance for the fusion of multispectral nightvision images. The method has a simple implementation and is computationally efficient[7].

3.2.2 Contents

To data, a variety of image fusion algorithms have been proposed. A popular class of algorithms are the multi-scale image fusion schemes, which decompose the source images into spatial primitives at multiple spatial scales, then integrate these primitives to form a new multi-scale transform-based representation, and finally apply an inverse multi-scale transform to reconstruct the fused image. However, most of these techniques are computationally expensive and tend to oversharpen edges, which makes them less suitable for application in multiscale schemes

Bilateral Filter: It can reverse the intensity gradient near sharp edges.

Guided Filter:

The two filtering conditions are:

- the local filter output is a linear transform of the guidance image *G*
- as similar as possible to the input image *I*.

The first condition implies that:

$$O_i = a_k G_i + b_k \qquad \forall i \in \omega_k$$

where the ω_k is a square window of size $(2r+1) \times (2r+1)$. **The local linear model ensures that the output image O has an edge only at locations where the guidance image G also has one.** Linear coefficients a_k and b_k are constant in ω_k. They can be

estimated by minimizing the squared difference between the output image O and the input image I in the window ω_k, i.e. minimizing the cost function E:

$$E(a_k, b_k) = \sum_{i \in \omega_k} \left((a_k G_i + b_k - I_i)^2 + \epsilon a_k^2 \right)$$

where ϵ_k is a regularization parameter penalizing large a_k. The coefficients a_k and b_k can directly be solved by linear regression. Since pixel i is contained in several different window ω_k, the value of O_i depends on the window over which it is calculated:

$$O_I = \overline{a}_i G_i + \overline{b}_i$$

The abrupt intensity changes in the guiding image G are still largely preserved in the output image O. The guided filter is a computationally efficient, edge-preserving operator which avoids the gradient reversal artefacts of the bilateral filter.

In Iterative guided filtering: In such a scheme the result G^{t+1} of the t-*th* iteration is obtained from the joint bilateral filtering of the input image I using the result G^t of the previous iteration step:

$$G_i^{t+1} = \frac{1}{K_i} \sum_{j \in \omega} I_j \cdot f(\parallel i - j \parallel) \cdot g(\parallel G_i^t - G_j^t \parallel)$$

Note that the initial guidance image G^1 can simply be a constant (e.g. zero) valued image since it updates to the Gaussian filtered input image in the first iteration step.

Proposed Method:

- Iterative guided filtering is applied to decompose the source images into base layers (representing large scale variations) and detail layers (containing small scale variations).
- Frequency-tuned filtering is used to generate saliency maps for the source images.
- Binary weighting maps are computed as the pixelwise maximum of the individual source saliency maps.
- Guided filtering is applied to each binary weighting map with its corresponding source as the guidance image to reduce noise and to restore spatial consistency.
- The fused image is computed as a weighted recombination of the individual source detail layers.

Visual saliency refers to the physical, bottom-up distinctness of image details. It is a relative property that depends on the degree to which a detail is visually distinct from its background. **Since saliency quantifies the relative visual importance of image details saliency maps are frequently used in the weighted recombination phase of multiscale image fusion schemes**. Frequency tuned filtering computes bottom-up saliency as local multiscale luminance contrast. The saliency map S for an image I is computed as

$$S(x, y) = \parallel I_\mu - I_f(x, y) \parallel$$

where

I_μ is the arithmetic mean image feature vector

I_f represents a Gaussian blurred version of the original image, using a 5 * 5 separable binomial kernel

$\| \cdot \|$ is the L_2 norm(Euclidian distance), and x, y are the pixel coordinates.

We compute saliency using frequency tuned filtering since a recent and extensive evaluation study comparing 13 state-of-the-art saliency models found that the output of this simple saliency model correlates more strongly with human visual perception than the output produced by any of the other available models.

Binary weight maps BW_{X_i} and BW_{Y_i} are then computed by taking the pixelwise maximum of corresponding saliency maps S_{X_i} and S_{Y_i}:

$$BW_{X_i}(x,y) = \begin{cases} 1 & \text{if } S_{X_i}(x,y) > S_{Y_i}(x,y) \\ 0 & \text{otherwise} \end{cases}$$

$$BW_{Y_i}(x,y) = \begin{cases} 1 & \text{if } S_{Y_i}(x,y) > S_{X_i}(x,y) \\ 0 & \text{otherwise} \end{cases}$$

The resulting binary weight maps are noisy and typically not well aligned with object boundaries, which may give rise to artefacts in the final fused image. Spatial consistency is therefore restored through guided filtering (GF) of these binary weight maps with the corresponding source layers as guidance images

$$W_{X_i} = GF(BW_{X_i}, X_i)$$
$$W_{Y_i} = GF(BW_{Y_i}, Y_i)$$

Fused detail layers are then computed as the normalized weighted mean of the corresponding source detail layers:

$$dF_i = \frac{W_{X_i} \cdot dX_i + W_{Y_i} \cdot dY_i}{W_{X_i} + W_{Y_i}}$$

The fused image F is finally obtained by adding the fused detail layers to the average value of the lowest resolution source layers:

$$F = \frac{X_3 + Y_3}{2} + \sum_{i=0}^{2} dF_i$$

By using guided filtering both in the decomposition stage and in the recombination stage, this proposed fusion scheme optimally benefits from both the multi-scale edge-preserving characteristics (in the iterative framework) and the structure restoring capabilities (through guidance by the original source images) of the guided filter. The method is easy to implement and computationally efficient.

3.2.3 Conclusion

We propose a multiscale image fusion scheme based on guided filtering. Iterative guided filtering is used to decompose the source images into base and detail layers. Initial binary weighting maps are computed as the pixelwise maximum of the individual

source saliency maps, obtained from frequency tuned filtering. Spatially consistent and smooth weighting maps are then obtained through guided filtering of the binary weighting maps with their corresponding source layers as guidance images. Saliency weighted recombination of the individual source detail layers and the mean of the lowest resolution source layers finally yields the fused image. The proposed multi-scale image fusion scheme achieves spatial consistency by using guided filtering both at the decomposition and at the recombination stage of the multiscale fusion process. Application to multiband visual (intensified) and thermal infrared imagery demonstrates that the proposed method obtains state-of-the-art performance for the fusion of multispectral nightvision images. The method has a simple implementation and is computationally efficient.

3.3 Image Fusion With Guided Filtering

3.3.1 Abstract

A fast and effective image fusion method is proposed for creating a highly informative fused image through merging multiple images. The proposed method is based on a two-scale decomposition of an image into a base layer containing large scale variations in intensity, and a detail layer capturing small scale details. A novel guided filtering-based weighted average technique is proposed to make full use of spatial consistency for fusion of the base and detail layers. Experimental results demonstrate that the proposed method can obtain state-of-the-art performance for fusion of multispectral, multifocus, multimodal, and multiexposure images.

3.3.2 Contents

Guided Filter

The filtering output O is linear transformation of the guidance image I in a local window ω_k centered at pixel k:

$$O_i = a_k I_i + b_k \qquad \forall i \in \omega_k$$

where ω_k is a square window of size $(2r+1) \times (2r+1)$. The linear coefficients a_k and b_k are contant in ω_k by minimizing the squared difference between the output image O and the input iamge P:

$$E(a_k, b_k) = \sum_{i \in \omega_k} \left((a_k I_i + b_k - P_i) + \epsilon a_k^2 \right)$$

where ϵ is a regularization parameter given by the user. The coefficients can be directly solved by linear regression as follows:

$$a_k = \frac{\frac{1}{|\omega|} \sum_{i \in \omega_k} I_i P_i - \mu_k \overline{P}_k}{\delta_k + \epsilon}$$
$$b_k = \overline{P}_k - a_k \mu_k$$

where μ_k and δ_k are the mean and variance of I in ω_k respectively. $|\omega|$ is the number of pixels in ω_k, and $\overline{P}_k$ is the mean of P in ω_k. Then the output image can be calculated according to above equation.

$$O_i = \overline{a}_i I_i + \overline{b}_i$$

where $\overline{a}_i = \frac{1}{|\omega|}\sum_{k\in\omega_i} a_k, \overline{b}_i = \frac{1}{|\omega|}\sum_{k\in\omega_i} b_k$.

The color image situation, the a_i and other calculators become vector version. See [4].

3.3.3 Fusion Frame

See figure 3.1.

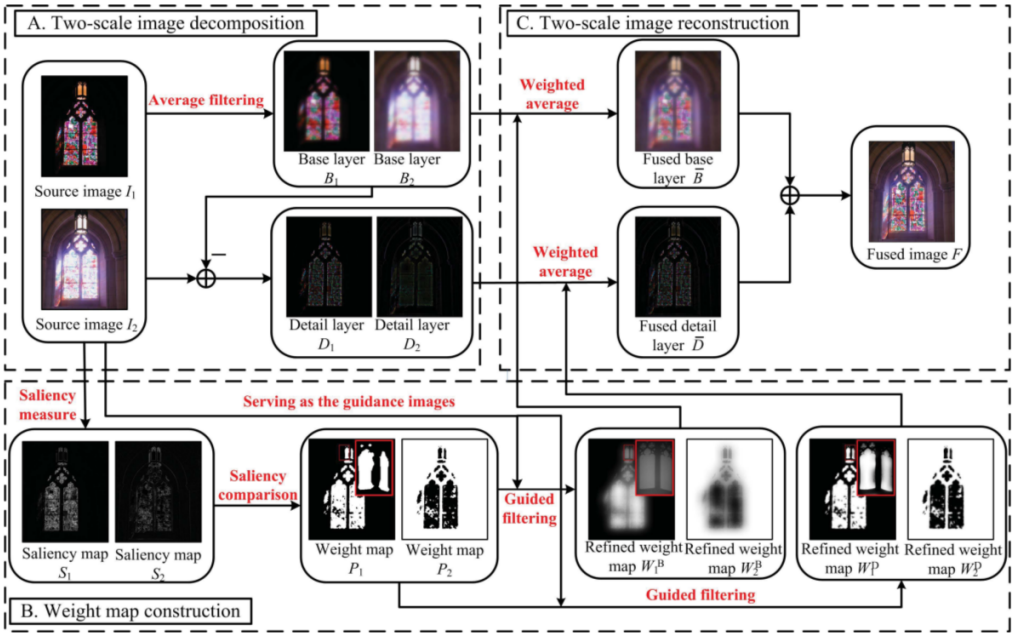

Fig. 3.1. Schematic diagram of the proposed image fusion method based on guided filtering.

3.3.4 Conclusion

Chapter 4

Saliency Detection

This chapter includes papers about saliency detection.

4.1 Frequency-tuned Salient Region Detection

4.1.1 Abstract

In this paper, we introduce a method for salient region detection that outputs full resolution saliency maps with well-defined boundaries of salient objects. These boundaries are preserved by retaining substantially more frequency content from the original image than other existing techniques. Our method exploits fea- tures of color and luminance, is simple to implement, and is computationally efficient. We compare our algorithm to five state-of-the-art salient region detection methods with a frequency domain analysis, ground truth, and a salient object segmentation application. Our method outperforms the five algorithms both on the ground-truth evaluation and on the segmentation task by achieving both higher precision and better recall.

4.1.2 Contents

Related work

Saliency estimation methods can broadly be classified as:

- biologically based
- purely computational
- combination of above two approaches

Itti base their method on the biologically plausible architecture proposed by Koch and Ullman. They determine center-surround contrast using a **Difference of Gaussians** (DoG). Frintrop presetn a method inspired by Itti's method, but they compute **center-surround differences** with square filters and use integral images to speed up the calculations.

Other methods are purely computational and are not based on biological vision principles. Ma and Zhang and Achanta et al. estimate saliency us- ing center-surround feature

distances. Hu et al. es- timate saliency by applying heuristic measures on initial saliency measures obtained by histogram thresholding of feature maps. Gao and Vasconcelos maximize the mu- tual information between the feature distributions of center and surround regions in an image, while Hou and Zhang rely on frequency domain processing.

The third category of methods are those that incorporate ideas that are partly based on biological models and partly on computational ones. For instance, Harel et al. create feature maps using Itti' s method but perform their normal- ization using a graph based approach. Other methods use a computational approach like maximization of information that represents a biologically plausible model of saliency detection.

Limitations

The saliency maps generated by most methods have low resolutioni. Itti's method produces saliency maps that are just $1/256^{th}$

Frequency-tuned Saliency Detection

DoG

DoG : Difference of Gaussians. DoG filter is widely used in edge detecion since it closely and efficiently approximates the Laplacian of Gaussian (LoG) filter, cited as the most satisfactory operator for detecting intensity changes when the standard deviations of the Gaussians are in the ratio $1 : 1.6$. The DoG has also been used for interest point detection and saliency detection. The DoG filter is given by :

$$\begin{aligned} DoG(x,y) &= \frac{1}{2\pi}\left[\frac{1}{\delta_1^2}e^{-\frac{x^2+y^2}{2\delta_1^2}} - \frac{1}{\delta_2^2}e^{-\frac{x^2+y^2}{2\delta_2^2}}\right] \\ &= G(x,y,\delta_1) - G(x,y,\delta_2) \end{aligned} \tag{4.1}$$

where δ_1 and δ_2 are the standard deviatioins of the Gaussian $(\delta_1 > \delta_2)$.

A DoG filter is a simple band-pass filter whose passband width is controlled by the ratio $\delta_1 : \delta_2$.

4.1.3 Conclusion

Chapter 5

Semantic SLAM

5.1 DeLS-3D: Deep Localization and Segmentation with a 2D Semantic Map

Chapter 6

MXNet

6.1 Optimizing Memory Consumption in DL

Over the last ten years, a constant trend in deep learning is towards deeper and larger networks. Despite rapid advances in hardware performance, cutting-edge deep learning models continue to push the limits of GPU RAM. So even today, it’ s always desirable to find ways to train larger models while consuming less memory. Doing so enables us to train faster, using larger batch sizes, and consequently achieving a higher GPU utilization rate.

6.1.1 Computation Graph

A computation graph describes the (data flow) dependencies between the operations in the deep network. The operations performed in the graph can be either fine-grained or coarse-grained.

The concept of a computation graph is explicitly encoded in packages like Theano and CGT. In other libraries, computation graphs appear implicitly as network configuration files. The major difference in these libraries comes down to how they calculate gradients. There are mainly two ways: performing back-propagation on the same graph or explicitly representing a backwards path to calculate the required gradients.

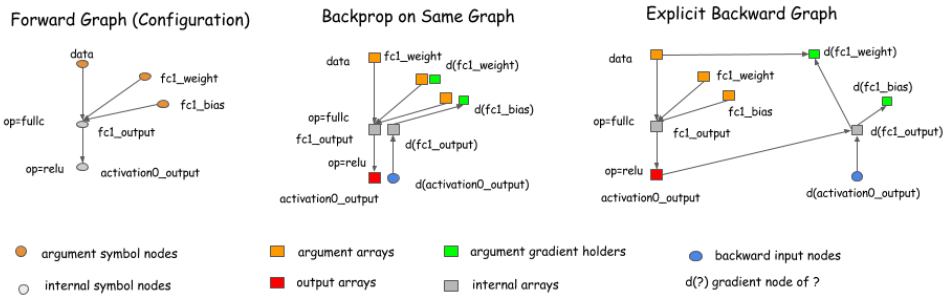

Fig. 6.1. The implicitly & explicitly back-propagation on Graph

Libraries like Caffe, CXXNet, and Torch take the former approach, performing back-prop on the original graph. Libraries like Theano and CGT take the latter approach, ex-

plicitly representing the backward path. In this discussion, we adopt the explicit backward path approach because it has several advantages for optimization.

We adopt the explicit backward path approach because it has several advantages for optimization.

Why is explicit backward path better? Two reasons:

- The explicit backward path clearly describes the dependency between computations. Like the following case, where we want to get the gradient of **A** and **B**. As we can see clearly from the graph, the computation of the $d(C)$ gradient doesn't depend on F. This means that we can free the memory of F right after the forward computation is done. Similarly, the memory of F can be recycled.

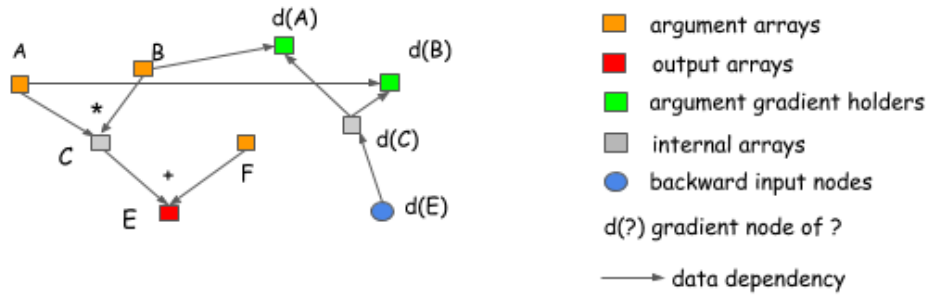

Fig. 6.2. Dependencies can be found quickly.

- Another advantage of the explicit backward path is the ability to have a different backward path, instead of a mirror of forward one.
  A common example is the split connectioncase, as shown in the following figure. In this example, the output of **B** is referenced by two operations. If we want to do

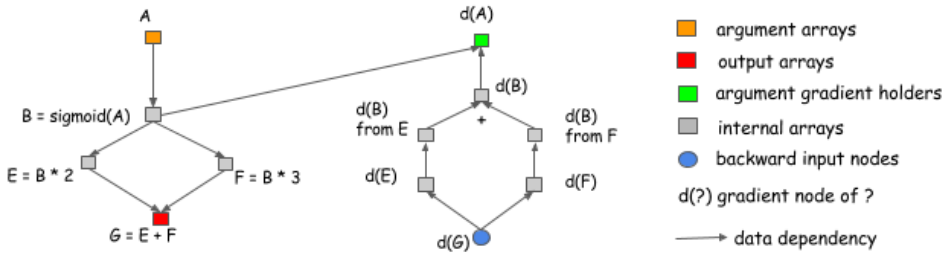

Fig. 6.3. Different backward path from forward path.

the gradient calculation in the same network, we need to introduce an explicit split layer. This means we need to do the split for the forward pass, too. In this figure, the forward pass doesn't contain a split layer, but the graph will automatically insert a gradient aggregation node before passing the gradient back to **B**. This helps us to save the memory cost of allocating the output of the split layer, and the operation cost of replicating the data in the forward pass.

6.1.2 What Can be Optimized?

As you can see, the computation graph is a useful way to discuss memory allocation optimization techniques. Already, we’ ve shown how you can save some memory by using the explicit backward graph. Now let’ s explore further optimizations, and see how we might determine reasonable baselines for benchmarking.

Assume that we want to build a neural network with n layers. Typically, when implementing a neural network, we need to allocate node space for both the output of each layer and the gradient values used during back-propagation. This means we need roughly $2n$ memory cells. We face the same requirement when using the explicit backward graph approach because the number of nodes in a backward pass is roughly the same as in a forward pass.

In-place Operations

One of the simplest techniques we can employ is *In-place memory sharing* across operations. For neural networks, we can usually apply this technique for the operations corresponding to activation functions.

"In-place" means using same memory for input and output. But you should be careful about that the result is used by more than one operation!

Standard Memory Sharing

In-place operations are not the only places where we can share memory. In the following example, because the value of B is no longer needed after we compute E, we can reuse B’ s memory to hold the result of E.

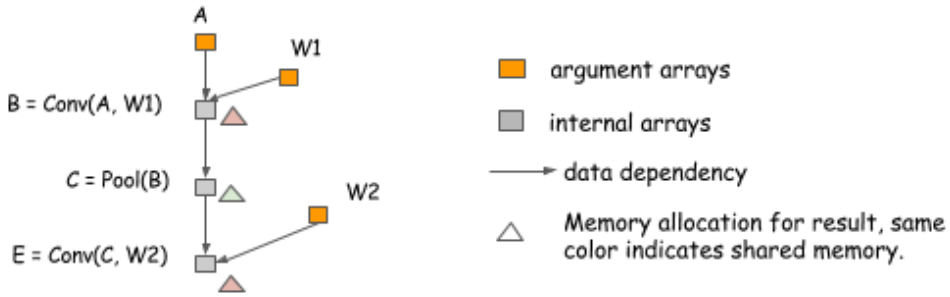

Fig. 6.4. Standard Memory sharing between **B** & the result of **E**.

Memory sharing doesn't necessarily require the same data shape. Note that in the preceding example, the shapes of **B** and **E** can differ. To handle such a situation, we can allocate a memory region of size equal to the maximum of that required by **B** and **E** and share it between them.

6.1.3 Memory Allocation Algorithm

Based on the " In-Place Operatioins", how can we allocate memory correctly?

The key problem is that we need to place resources so that they don’t conflict with each other. More specifically, each variable has a **life time** between the time it gets computed

until the last time it is used. In the case of the multi-layer perceptron, the life time of *fc1* ends after *act1* get computed. See below figure:

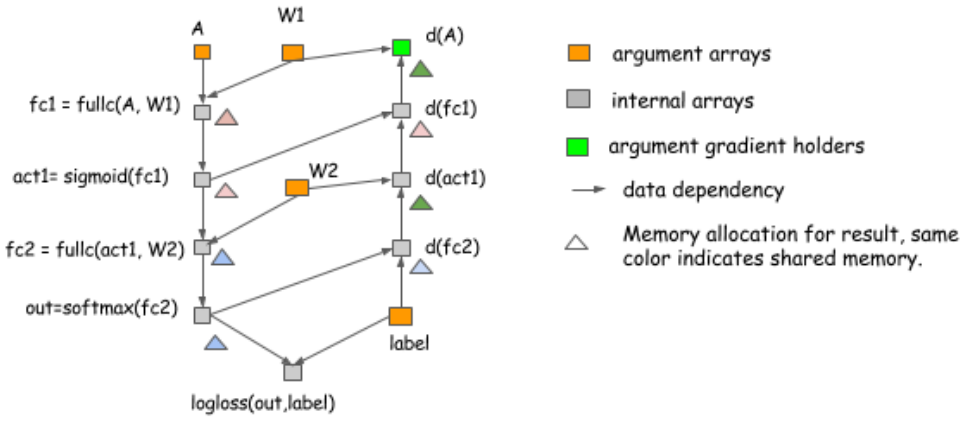

Fig. 6.5. Standard Memory sharing between **B** & the result of **E**.

The principle is to allow memory sharing only between variables whose lifetimes don't overlap. There are multiple ways to do this. You can construct the conflicting graph with each variable as a node and link the edge between variables with overlapping lifespans, and then run a graph-coloring algorithm. This likely has $O(n^2)$ complexity, where n is the number of nodes in the graph. This might be too costly.

Let's consider another simple heuristic. The idea is to simulate the procedure of traversing the graph, and keep a count of future operations that depends on the node.

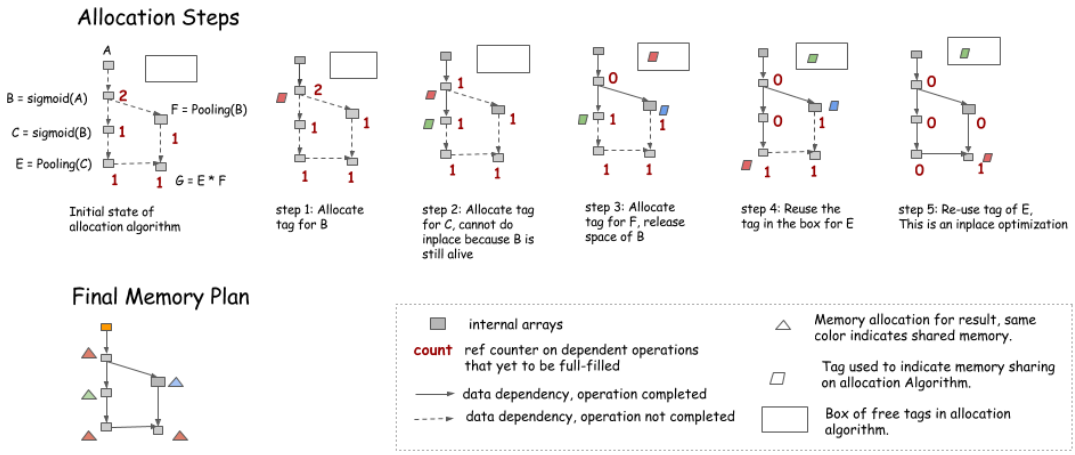

Fig. 6.6. Standard Memory sharing between **B** & the result of **E**.

- An in-place optimization can be performed when only the current operation depends on the source (i.e. count == 1).
- Memory can be recycled into the box on the upper right corner when the *count* goes to 0.
- When we need new memory, we can either get it from the box or allocate a new one.

Noet: During the simulation, no memory is allocated. Instead, we keep a record of how much memory each node needs, and allocate the maximum of the shared parts in the final memory plan.

6.1.4 Static vs. Dynamic Allocation

The major difference is that static allocation is only done once, so we can afford to use more complicated algorithms. For example, we can search for memory sizes that are similar to the required memory block. The Allocation can also be made graph aware. We’ll talk about that in the next section. Dynamic allocation puts more pressure on fast memory allocation and garbage collection.

There is also one takeaway for users who want to rely on dynamic memory allocations: do not unnecessarily reference objects. For example, if we organize all of the nodes in a list and store then in a Net object, these nodes will never get dereferenced, and we gain no space. Unfortunately, this is a common way to organize code.

6.1.5 Memory Allocation for Parallel Operations

In the previous section, we discussed how we can simulate running the procedure for a computation graph to get a static allocation plan. However, optimizing for parallel computation presents other challenges because resource sharing and parallelization are on the two ends of a balance. Let’s look at the following two allocation plans for the same graph:

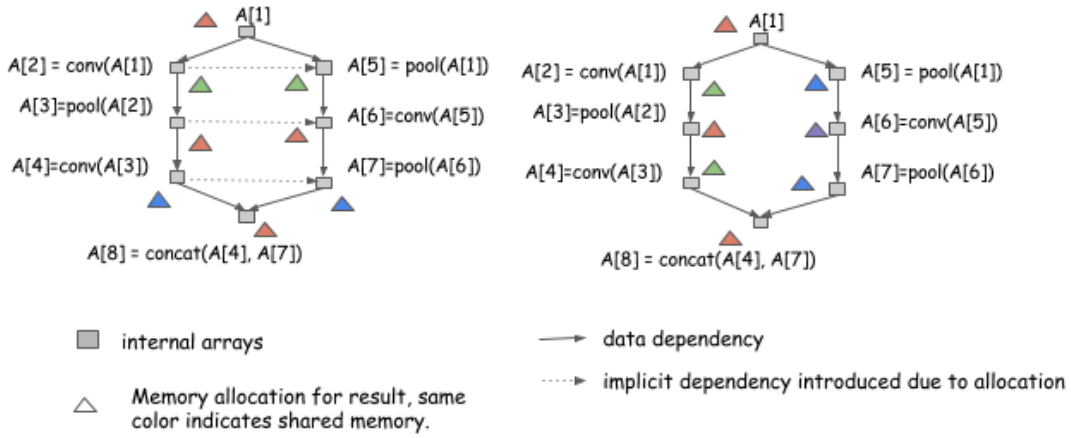

Fig. 6.7. Standard Memory sharing between **B** & the result of **E**.

Both allocation plans are valid if we run the computation serially, **from** $A[1]$ **to** $A[8]$**.** However, the allocation plan on the left introduces additional dependencies, which means we can't run computation of $A[2]$ and $A[5]$ in parallel. The plan on the right can. To parallelize computation, we need to take greater care.

Be Correct and Safe First

Being correct is our first principle. This means to execute in a way that takes implicit dependency memory sharing into consideration. You can do this by adding the implicit

dependency edge to the execution graph. Or, even simpler, if the execution engine is mutation aware, as described in our discussion of dependency engine design, push the operation in sequence and write to the same variable tag that represents the same memory region.

Always produce a safe memory allocation plan. This means never allocate the same memory to nodes that can be parallelized. This might not be ideal when memory reduction is more desirable, and we don't gain too much when we can get benefit from multiple computing streams simultaneously executing on the same GPU.

Try to Allow More Parallelization

Now we can safely perform some optimizations. The general idea is to try and encourage memory sharing between nodes that can't be parallelized. You can do this by creating an ancestor relationship graph and querying it during allocation, which costs approximately $O(n^2)$ in time to construct. We can also use a heuristic here, for example, color the path in the graph. As shown in the following figure, when you try to find the longest paths in the graph, color them the same color and continue.

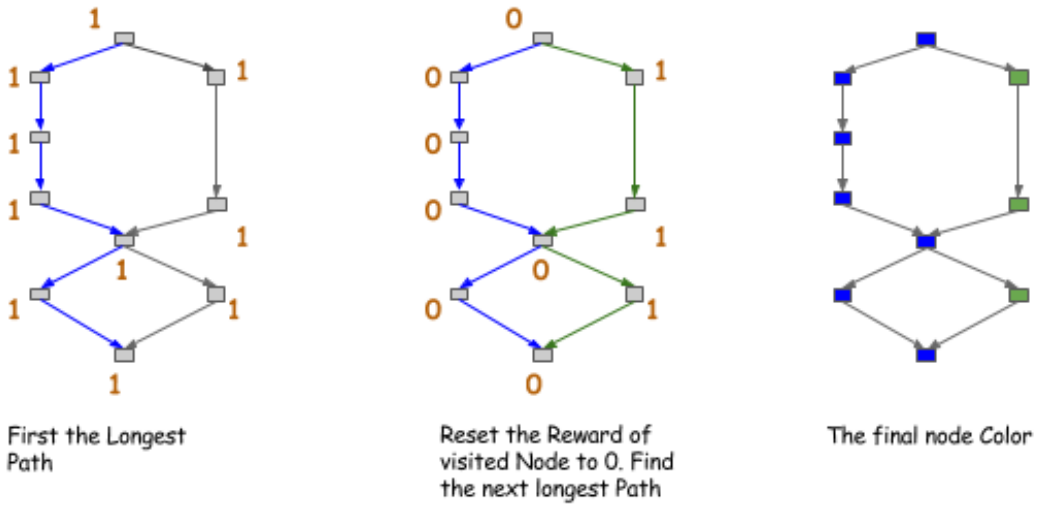

Fig. 6.8. Color the longest paths in the Graph.

After you get the color of the node, you allow sharing (or encourage sharing) only between nodes of the same color. This is a stricter version of the ancestor relationship, but is costs only $O(n)$ of time if you search for only the first k path.

6.1.6 How Much Can we Save ?

On coarse-grained operation graphs that are already optimized for big operations, you can reduce memory consumption roughly by half. You can reduce memory usage even more if you are optimizing a fine-grained computation network used by symbolic libraries, such as Theano.

6.1.7 References

More details can be found in: Opimizing the Memory Consumption in DL(MXNet).

6.2 Deep Learning Programming Style

Two of the most important high-level design decisions

- Whether to embrace the symbolic or imperative paradigm for mathematical computation
- Whether to build networks with bigger or more atomic operations

6.2.1 Symbolic vs. Imperative Program

即：符号式编程 vs. 命令式编程

Symbolic programs are a bit different. With symbolic-style programs, we first define a (potentially complex) function abstractly. When defining the function, no actual numerical computation takes place. We define the abstract function in terms of **placeholder values**(占位符). Then we can compile the function, and evaluate it given real inputs.

This operation generates a computation graph (also called a symbolic graph) that represents the computation.

Most symbolic-style programs contain, either explicitly or implicitly, a compile step.

真正的计算只发生在传入数值之时，在这之前，都没有任何计算发生。

The defining characteristic of symbolic programs is their clear separation between building the computation graph and executing it.For neural networks, we typically define the entire model as a single compute graph.

6.2.2 Imperative Programs Tend to be More Flexible

未完待续...

Chapter 7

Tips in DL

7.1 Enlarge the FOV

增加网络的感受野。目前看到的主要方法如下：

- CRFs[6]
- Global Graph-reasoning module[2]
- Pooling
- Dilated conv[8]
-

7.2 Upsampling

在卷积以及 Pooling 之后保持分辨率。目前看到的主要方法如下：

- Padding
- Deconvolution
- Uppooling
-

7.3 Multiscale Ability

在目标检测 (Object Detection) 中加入多尺度信息。记得的有以下几个方法：

- Pyramid Network
- Stacked CNN ?
-

7.4 Dilated Convolution

主要原理如下。

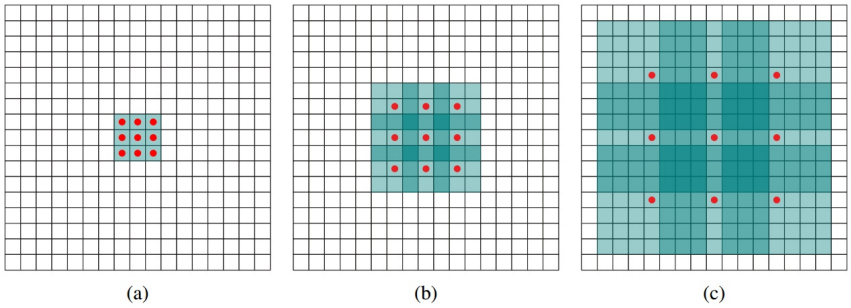

Fig. 7.1. Dilated Convolution 示意图

注意，下文提到的 N-Dilated Conv 中的 $N = 1, 2, 3, \ldots$ 是指图中相邻红点之间的间隔。

图7.1中，(a) 图对应 3x3 的 1-dilated conv，和普通的卷积操作一样，(b) 图对应 3x3 的 2-dilated conv，实际的卷积 kernel size 还是 3x3，但是空洞为 1，也就是对于一个 7x7 的图像 patch，只有 9 个红色的点和 3x3 的 kernel 发生卷积操作，其余的点略过。也可以理解为 kernel 的 size 为 7x7，但是只有图中的 9 个点的权重不为 0，其余都为 0。可以看到虽然 kernel size 只有 3x3，但是这个卷积的感受野已经增大到了 7x7（**如果考虑到这个 2-dilated conv 的前一层是一个 1-dilated conv 的话，那么每个红点就是 1-dilated 的卷积输出，所以感受野为 3x3，所以 1-dilated 和 2-dilated 合起来就能达到 7x7 的 conv**）,(c) 图是 4-dilated conv 操作，同理跟在两个 1-dilated 和 2-dilated conv 的后面，能达到 15x15 的感受野。对比传统的 conv 操作，3 层 3x3 的卷积加起来，stride 为 1 的话，只能达到 (kernel-1)*layer+1=7 的感受野，也就是和层数 layer 成线性关系，而 dilated conv 的感受野是指数级的增长。

Dilated 的好处是不做 Pooling 算是信息的情况下，加大了感受野，让每个卷积核输出都包含较大范围的信息。在图像需要全局信息或者语音文本需要较长的 Sequence 信息依赖的问题中，都能很好的应用 Dilated Convolution, 比如图像分割、语音合成 WaveNet、机器翻译 ByteNet。

WaveNet 的例子。

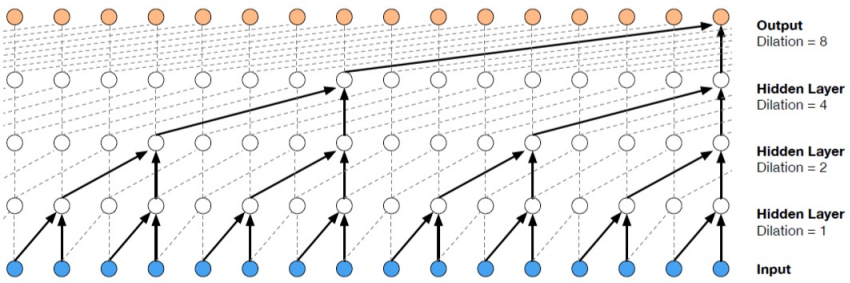

Fig. 7.2. Dilated Convolution 在 WaveNet 中的应用示意图

参考文献：Dilated Conv 知乎

7.5 Deconvolutional Network

参考文献：Deconvolution Networks
可能应用的领域：Visualization, Pixel-wise Prediction, Unsupervised Learning, Image Generation.
大致可分为以下几个方面：

- Unsupervised Learning

  其实是 Covolutional Sparse Coding. 这里的 Deconv 只是观念上和传统的 Conv 反向，传统的 conv 是从图片生成 feature map，而 deconv 是用 unsupervised 的方法找到一组 kernel 和 feature map，让它们重建图片。

- CNN Visualization

  通过 deconv 将 CNN 中 conv 得到的 feature map 还原到像素空间，以观察特定的 feature map 对哪些 pattern 的图片敏感，这里的 deconv 其实不是 conv 的可逆运算，只是 conv 的 transpose，所以 tensorflow 里一般取名叫 transpose_conv。

- Upsampling

  在 pixel-wise prediction 比如 image segmentation[4] 以及 image generation[5] 中，由于需要做原始图片尺寸空间的预测，而卷积由于 stride 往往会降低图片 size，所以往往需要通过 upsampling 的方法来还原到原始图片尺寸，deconv 就充当了一个 upsampling 的角色。

下面主要介绍这三个方面的论文。

7.5.1 Convolutional Spare Coding

第一篇：Deconvolutional Netwoks
主要用于学习图片的中低层级的特征表示，属于 Unspervised Feature Learning。
更多内容参考本小节的参考文献。

7.5.2 CNN 可视化

ZF-Net 中利用 Deconv 来做可视化，它是将 CNN 学习到的 Feature Map 的卷积核，取转置，将图片特征从 Feature Map 空间转化到 Pixel 空间，用于发现哪些 Pixel 激活了特定的 Feature Map，达到分析理解 CNN 的目的。

7.5.3 Upsampling

用于 FCN[5] 和 DCGAN。

7.6 Dilated Network 与 Deconv Network 之间的区别

Dilated Convolution 主要用于增加感受野，而不是 Upsampling；Deconv Network 主要用于 Upsample，即增加图像分辨率。
对于标准的 $k \times k$ 的卷及操作，stride 为 s，分为一下几种情况：

- $s > 1$

  即卷积的同时做了降采样，输入 Feature Map 的分辨率[1]下降。但这一般也会增加感受野。

- $s = 1$

  普通的步长为 1 的卷积，输入与输出分辨率相同。

- **0 < s < 1**

  Fractionally strided convolution. 相当于图像做 upsampling。比如 $s = 0.5$ 时，意味着图像像素之间 padding 一个空白的像素 (像素值为 0) 后，stride 改为 1 进行卷积，达到一次卷积看到的空间范围变大的目的。

7.7 Uppooling

In the convnet, the max pooling operation is non-invertible, however we can obtain an approximate inverse by recording the locations of the maxima within each pooling region in a set of switch variables. In the deconvnet, the unpooling operation uses these switches to place the reconstructions from the layer above into appropriate locations, preserving the structure of the stimulus.

也就是说用一组开关变量保存最大值在 Pooling Region 中的位置。

参考文献：Quora Answer

7.8 目标检测中的 mAP 的含义

- 对于类别 C, 在一张图像上

  首先计算 C 在一张图像上的精度。

$$Precision_C = \frac{N(TP)_C}{N(Total)_C}$$

  其中，$Precision_C$ 为类别 C 在一张图像上的精度。$N(TP)_C$ 为算法检测正确 (True Positive) 的 C 的个数，检测是否正确按照 $IoU > 0.5$ 算，同理，$T(Total)_C$ 为这一张图像所有 C 类的个数。所以则一步，仅涉及一个类别 C 以及一张图像。

- 对于类别 C，在多张图像上

  这一步计算的是类别 C 的 AP 指数。

$$AveragePrecision_C = \frac{\sum Precision_C}{N(TotalImage)_C}$$

  其中，$AveragePrecision_C$ 是类别 C 的 AP 指数，$Precision_C$ 为上文计算得到的类别 C 的在一张图像上的精度，然后对所有包含类别 C 的图像上的 C

[1]分辨率是指像素的多少，而尺度是指模糊程度的大小，即 Gaussian Filter 中的方差 δ

的精度 $Precision_C$ 求和；$N(TotalImage)_C$ 为包含类别 C 的图像的数量，也对应于分子中求和所涉及的图像。

- 在整个数据集上，多个类别

  mAP 在上一步的计算结果的基础上，计算所有类别的 AP 和 / 总的类别数。

$$meanAveragePrecision = \frac{\sum_C AveragePrecision_C}{N(Class)}$$

也就是相当于计算所有类别的 **AP** 的平均值，是对应于类别总数的平均值。

参考文献：知乎文章

7.9 统计学习方法

一个比较好的总结：机器学习常见算法个人总结

7.10 待续

References

[1] Shuichi Asano, Tsutomu Maruyama, and Yoshiki Yamaguchi. Performance comparison of fpga, gpu and cpu in image processing. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 126--131. IEEE, 2009.

[2] Xinlei Chen, Li-Jia Li, Li Fei-Fei, and Abhinav Gupta. Iterative visual reasoning beyond convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[3] Chao Li, Yanjing Bi, Franck Marzani, and Fan Yang. Fast fpga prototyping for real-time image processing with very high-level synthesis. *Journal of Real-Time Image Processing*, pages 1--18, 2017.

[4] Shutao Li, Xudong Kang, and Jianwen Hu. Image fusion with guided filtering. *IEEE Transactions on Image Processing*, 22(7):2864--2875, 2013.

[5] J. Long, E. Shelhamer, and T. Darrell. Fully Convolutional Networks for Semantic Segmentation. *ArXiv e-prints*, November 2014.

[6] Roberto Cipolla Marvin T.T. Teichmann. Convolutional crfs for semantic segmentation. *arXiv:https://arxiv.org/pdf/1805.04777.pdf*, 2018.

[7] Alexander Toet and Maarten A Hogervorst. Multiscale image fusion through guided filtering. In *SPIE Security+ Defence*, pages 99970J--99970J. International Society for Optics and Photonics, 2016.

[8] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. In *ICLR*, 2016.

Index