Trabalho Prático II

Guilherme Krzisch e Matheus Cavalca
Sistemas Embarcados

June 4, 2015

1 Contexto

O objetivo deste trabalho é paralelizar uma aplicação (ou algoritmo) responsável por aplicar um filtro gaussiano em uma imagem, utilizando o sistema operacional HellFireOS Lite e o simulador MPSoC. A imagem deve ser em grayscale, e não deve ser muito grande, para evitar muito tempo de simulação.

2 Paralelização do algoritmo

Para paralelizar o processamento iremos utilizar o modelo mestre-escravo, dividindo em 9 processadores, formando um MPSoC 3x3. Haverá um mestre para coordenar as tarefas e oito escravos responsáveis pelo processamento em si.

O mestre é responsável por dividir a imagem em partes e enviar para os escravos. Todos os oito escravos funcionam do mesmo jeito: possuem uma tarefa que fica esperando receber uma parte da imagem do mestre. Quando recebem, aplicam o filtro gaussiano (algoritmo fornecido) e devolvem a imagem processada para o mestre; após, voltam a esperar o recebimento de mais tarefas (em um loop, ou seja, eles teoricamente podem receber um número variado de tarefas. Para o presente trabalho, cada escravo recebe exatamente duas tarefas).

O mestre possui dois tipos de tarefas distintas: (1)para receber as tarefas dos escravos ele cria uma tarefa responsável por enviar as partes das imagens para um determinado escravo, e após receber as respostas desse mesmo escravo. Isso é necessário para termos um controle sobre qual parte da imagem estamos processando naquele momento. Ao receber a imagem processada, ele guarda em uma variável global que será a imagem final (2) Tarefa principal responsável por fazer a sincronização entre os processos/tasks. Utilizando um semáforo, inicializado com 0, ele espera (HF_SemWait()) N vezes, onde N é o número de partes em que a imagem foi dividida. Após N recebimentos, temos a imagem final na variável global que os escravos popularam. A tarefa então imprime na tela os pixels da imagem com o filtro aplicado.

Para o presente trabalho utilizamos uma imagem com tamanho 128 x 128 pixels. Portanto, dividimos a imagem em 16 partes de 32 x 32. Com 8 escravos, o mestre terá 8 tarefas (do tipo (1)) que enviam duas partes de 32 x 32 para cada

escravo, e esperam a resposta. A seguir há a correspondência entre escravos e a parte da imagem a ser processada.

Escravo 1 (32x32)	Escravo 2	Escravo 3	Escravo 4
Escravo 5	Escravo 6	Escravo 7	Escravo 8
Escravo 1	Escravo 2	Escravo 3	Escravo 4
Escravo 5	Escravo 6	Escravo 7	Escravo 8

2.1 Alterações necessárias

Tivemos que fazer algumas alterações nas configurações dos makefiles do HellFireOS e do simulador MPSoC para o correto funcionamento.

No Makefile do HellFireOS adicionamos uma configuração para utilizar um NOC 3x3, com um total de tarefas igual a 11 para cada processador (o mestre possui 9 tarefas, mais duas padrão):

```
MY_COREX_OS_CONFIG = -DDEBUG -DMAX_TASKS=11 -
DHEAP_SIZE=524288 -DSTACK_MAGIC=0xa5a5a5a5 -DCPU_SPEED=25000000
-DTICK_TIME=18 -DN_CORES=9 -DNOC_INTERCONNECT
-DNOC_WIDTH=3 -DNOC_HEIGHT=3 -DPACKET_SIZE=64
-DFAST_STDOUT #-DCOUNTER_REG=COUNTER
```

Além disso adicionamos mais uma lista de cores, para conter nove entradas:

```
MANY_CORE_LIST3 = 0 1 2 3 4 5 6 7 8
```

E, por fim, adicionamos uma entrada chamada "filter_test" para gerar os arquivos necessários para a execução (nos baseamos na entrada do "noc_test").

Também modificamos o Makefile do MPSoC para funcionar com um NOC 3x3:

```
noc_3x3: $(GCC) -o mpsoc_sim ./source/mpsoc_sim.c ./source/noc.c
-lm -DN_CORES=9 -DNOC_WIDTH=3 -DNOC_HEIGHT=3 -
DNOC_BUFFER_SIZE=16 -DOS_PACKET_SIZE=64
```

3 Resultados

Utilizando a figura 1 como entrada (128 x 128, grayscale), obtemos a figura 2 como saída do algoritmo. Podemos comparar com a figura 3, que é a saída do algoritmo não paralelizado.

Figure 1: Original

Figure 2: Saída do algoritmo paralelo

Figure 3: Saída do algoritmo não paralelo

Pode-se observar as linhas em formato de "grade" dividindo a figura, em exatamento 16 partes. Isso é o esperado com o que foi implementado, uma vez que só enviamos uma matriz 32x32 para os escravos, e para calcular os pixels das bordas da figura (pixels a uma distância de até 3 pixels da borda), precisaríamos de pixels além dos contidos na matriz 32x32.

Para melhorar esse resultado, pode-se aplicar várias técnicas. Poderíamos enviar sempre mais pixels para os escravos poderem fazer um processamento completo; porém isso traria uma maior complexidade na implementação, uma vez que isso depende da parte a ser enviada; o algoritmo deixaria de ser genérico. Outra alternativa, mais simples, é a de copiar os pixels mais próximos em que foi aplicado o filtro gaussiano para os pixels onde não foi possível. Por exemplo, pixels da borda superior da imagem receberiam o mesmo valor do primeiro pixel abaixo deles que foi processado (exemplo: o pixel em que x = 0 e y = 10 receberia o mesmo conteúdo do pixel x = 3 e y = 10). Implementamos essa solução por ser simples de integrar no código que já tínhamos, se mostrando relativamente eficaz e melhorando a imagem gerada, como pode ser observado na figura 4.

Figure 4: Saída do algoritmo paralelo utilizando melhorias

O tempo de processamento (em ciclos de relógio) para o algoritmo paralelo foi de aproximadamente 14678158. O tempo de processamento em cada escravo de cada parte recebida foi em média de 3758366. O tempo para o algoritmo original foi de 59971128[1]

[1] Notebook, core i5, Ubuntu 15.04

4 Conclusão

Acreditamos que conseguimos um bom resultado com a paralelização do algoritmo. O tempo de processamento caiu de 60M ciclos (versão não paralela) para cerca de 15M ciclos na versão paralelizada, diminuindo o tempo em quatro vezes. A imagem gerada é parecida com a original. Para trabalhos futuros poderíamos melhorar a saída da imagem, utilizando a técnica descrita na seção 3, eliminando as "quebras em grid" da imagem gerada.