Arquiteturas Ortovetoriais

Geraldo Lino de Campos

Escola Politécnica da Universidade de São Paulo,
Departamento de Engenharia de Computação e Sistemas Digitais
Caixa Postal 8174 - São Paulo 01065-970 - BRASIL
Tel + 55 11 818 5288 - FAX + 55 11 818 5718
e-mail: RTC@FPSP.FAPESP.BR

Resumo

Processadores vetoriais emitem instruções que executam uma mesma operação individual sobre os elementos de um vetor, correspondente ao código gerado por uma mesma instrução do programa fonte. Esta estratégia apresenta o inconveniente de que uma ampla classe de construções não podem ser vetorizadas. Processadores baseados na arquitetura proposta neste artigo emitem um conjunto de instruções operando sobre todas as variáveis envolvidas numa malha mais interna, e conseguem processar uma classe bem mais ampla de comandos, incluindo os que apresentam recorrências e condicionais, com a mesma eficiência com que os processadores vetoriais executam códigos vetorizáveis.

Abstract

Vectorial processors operate issuing instructions that execute an individual operation over a set of elements, all pertaining to the code generated by a single statement of the source language. Its main drawback is that a large class of constructs can not be vectorized, specially those including recurrences and conditionals. Processors built upon the proposed orthovectorial architecture issue different instructions over a set of operands generated by all the statements present in an inner loop, and are able to process complex loops, even with recurrences and conditionals, with the same efficiency vectorial processors operate on vectorizable code.

1 Introdução

Processadores vetoriais são dispositivos projetados para superar dois problemas encontrados nas arquiteturas convencionais:

1 - Taxas de leitura, decodificação e emissão de instruções, uma vez que é bastante difícil realizar essas operações para mais do que algumas instruções por ciclo; é o chamado "gargalo de Flynn [Fly66].

2 - O tempo de ciclo. Em princípio, o tempo de ciclo pode ser reduzido arbitrariamente pelo aumento no número de estágio do pipeline, mas isto aumenta as dependências de dados e de controle, reduzindo os benefícios de sua utilização.

Uma análise dos processadores vetoriais mostra como isto é conseguido:

O gargalo de Flynn é evitado pela utilização de um processador separado para o cálculo dos endereços dos elementos individuais dos vetores, como se vê na figura 1. Este processador será chamado neste texto de Unidade de Endereços.

O pipeline pode ser feito com mais estágios, uma vez que as operações não dependem entre si, já que apenas operações independentes de dados são vetorizadas. Isto limita a percentagem de código que pode ser vetorizado.

Este texto apresenta uma nova abordagem para eliminar os problemas 1 e 2 apresentados acima. A principal diferença é encarar as malhas mais internas como unidade básica de trabalho em lugar de operações individuais. Isto permite que uma classe muito maior de construções sintáticas sejam executadas no modo "vetorizado".

A principal diferença reside na Unidade de Endereços, que deve ser capaz de gerar seqüencialmente os endereços de todos os elementos de vetores utilizados em uma malha mais interna, na mesma ordem em que seriam gerados pela execução de um programa convencional, e de forma independente do fluxo de controle principal. A figura 2 apresenta esta nova Unidade de Endereços, onde se destaca a fila de entrada, essencial para permitir essa operação assíncrona. Uma unidade análoga, porém sem a fila de entrada, é necessária para as operações de escrita à memória.

Toda arquitetura que obedecer ao princípio básico acima pode ser chamada de ortovetorial, uma vez que opera de maneira semelhante à vetorial, porém seleciona seus operandos de forma ortogonal a ela. A descrição apresentada a seguir representa apenas uma possível implementação, que procura aproveitar suas vantagens e contornar seus possíveis inconvenientes.

Para realizar acessos à fila de entrada por todas as instruções, reserva-se o número de um dos registradores de ponto flutuante para designar a cabeça da fila de entrada; subentende-se que a leitura desse registrador causará uma atualização automática da fila, e que se não houver um elemento presente a instrução ficará bloqueada até que um elemento venha da memória.

Escrever nesse registrador faz com que a unidade de endereçamento de escrita gere o próximo endereço de memória, onde será escrito o valor movido para esse registrador.

Para exemplos, este texto utilizará o processador DLXV, utilizado num livro texto de arquitetura bastante

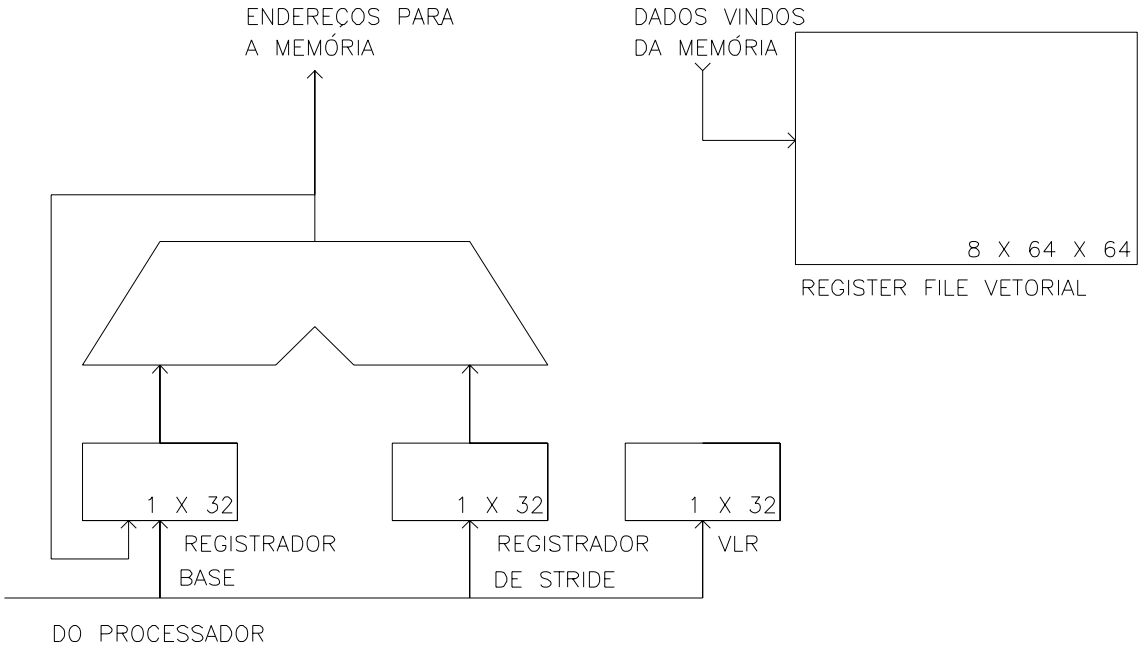

Figura 1 - Unidade de Endereços em uma máquina vetorial

VLR é o registrador que contém o comprimento do vetor a carregar (Vector Lenght Register). Os números apresentam valores típicos para o tamanho dos registradores.

popular [Hen90], adotando seus nomes de operações e de registradores.

O registrador que representa a cabeça da fila será chamado de DAR (Data Access Register).

O uso de filas de entrada está associado com arquiteturas desacopladas [Smi84], e é parcialmente utilizado em outras arquiteturas, como a WM [Wul88], mas não é considerado um elemento central, como o é para a Arquitetura Ortovetorial.

A seção 2 apresenta os mecanismos básicos de geração de endereços e as extensões necessárias a uma arquitetura típica para que se possa obter bons resultados com a transição para uma arquitetura ortovetorial, e compara os recursos necessários com os de um processador vetorial equivalente. A seção 3 estende o modelo para um caso mais realista, exibindo a ortovetorização de alguns núcleos não vetorizáveis dos Lawrence Livermore Kernels (LLKs) [Mah86], e considera o caso superescalar. A seção 4 apresenta resultados completos para os LLKs, e a seção 5 apresenta as conclusões.

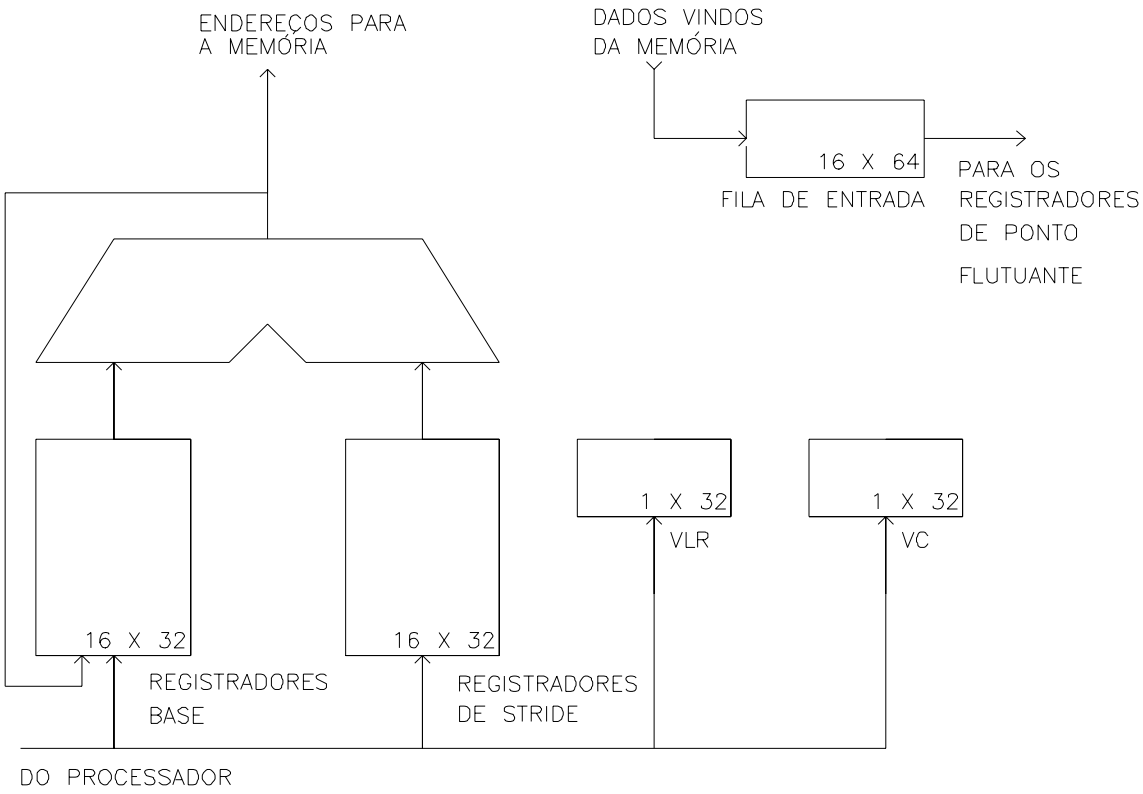

Figura 2 - Unidade de Endereços em uma máquina ortovetorial

VLR contém agora o número de iterações da malha. Em cada iteração um endereço de memória é gerado em correspondência com cada par de valores contidos nos conjuntos de registradores de base e de stride. O número de entradas válidas nesses conjuntos de registradores é contido em VC. Os números apresentam valores típicos para o tamanho dos registradores.

2 Mecanismos Básicos

Para obter a maior simplicidade possível na apresentação dos conceitos, esta seção supõe que o subsistema de memória seja perfeito, e capaz de suportar uma operação de leitura e uma operação de escrita por ciclo do processador, sem latência; na próxima seção estas hipóteses simplificadoras serão levantadas.

Adota-se uma latência de 2 para que as unidades de soma e multiplicação em ponto flutuante completem sua operação. Esta hipótese não é fundamental; os resultados desta seção se mantêm se este número for maior, mas os exemplos se tornarão mais extensos.

2.1– Operação das unidades de endereço

A operação da unidade de endereço para leitura é bastante simples: antes de iniciar uma operação ortovetorial deve-se ter instruções que inicializem tantos pares de valores dos registradores de endereço e de stride quantos sejam os vetores referidos na malha mais interna, e a seguir o "vector length register", VLR, com o número de iterações a realizar.

A instrução

```
MOV2LA R1,R2
```

move o valor base de um vetor, presente em R1, e o respectivo stride, contido em R2, para o próximo par de registradores de base e de stride disponíveis na unidade de endereço de leitura. Uma instrução equivalente, MOV2SA, envia valores para a unidade de endereços de escrita, e a instrução

```
MOVI2S VLR, R1
```

move o "comprimento vetor", isto é, o número de iterações, para o VLR. É importante lembrar que existe um único VLR, que é utilizado pela unidade de endereços de leitura; uma vez inicializada, a unidade de endereços de escrita opera indefinidamente, enquanto valores forem escritos no DAR.

A escrita de um valor em VLR inicia a operação autônoma da unidade de endereços de leitura, que pode para momentaneamente se a fila de entrada estiver cheia, e para definitivamente após leitura do número de elementos especificado.

Para a geração do primeiro endereço, o número contido no primeiro registrador de base é enviado à memória, e ao mesmo tempo é somado ao valor contido no registrador de stride correspondente; o resultado substitui o valor contido no registrador base. Os outros registradores são processados da mesma forma, até que todos os registradores que foram inicializados tenham sido processados; o valor de VLR é decrementado, e a seqüência é repetida até que seu valor atinja zero.

As operações de escrita à memória seguem o mesmo padrão, com a diferença que os endereços de escrita são gerados apenas quando uma palavra é escrita em DAR.

2.2– Um exemplo simples

Os mecanismos básicos podem ser apresentados estudando-se um exemplo simples, o tradicional DAX-PY:

```
Y = a * X + Y
```

onde X e Y são vetores e a é um escalar.

O código para inicialização da unidade de endereços de leitura será

```
LDI     R3,#número de elementos
LDI     R1,#endereço de X
LDI     R2,#stride de X
MOV2LA  R1,R2
LDI     R1,#endereço de Y
LDI     R2,#stride de Y
MOV2LA  R1,R2
```

para a unidade de endereços de escrita, será

```
MOV2SA  R1,R2
```

(lembre-se que R1 e R2 ainda contêm a base e o stride do vetor Y), e para o início da operação da unidade de leitura de endereços,

```
MOVI2S  VLR,R3
```

Embora aparentemente extenso, este código se compara bem com o código de "strip mining" necessário nos processadores vetoriais, com a vantagem adicional de ser utilizado apenas uma vez.

O código para a malha é extremamente simples:

```
        LD      F0,a        ; (1)carrega o escalar a
LOOP    MULTD   F2,F0,DAR   ; (2)
        ADDD    DAR,F2,DAR  ; (3)
        SUBI    R3,R3,#1    ; (4)
        BNZ     R3,LOOP     ; (5) verifica se acabou
```

A linha (2) multiplica o escalar a pelo próximo elemento de X; a linha (3) soma o resultado ao próximo elemento de Y, e envia o resultado para o próximo elemento de Y.

A latência da operação MULTD causará um ciclo de parada (stall); neste caso, o simples escalonamento das instruções pode resolver o problema, mas no caso mais geral uma técnica mais sofisticada, como pipeline em software [Lam88] teria de ser utilizada, causando um overhead de inicialização e finalização. Na seção 3 será apresentada uma solução por hardware que permite ocultar a latência sem esses problemas.

Escalonando a malha para utilizar o delay slot, temos o código final:

```
        LD      F0,a
LOOP    SUBI        R3,R3,#1
        MULTD   F2,F0,DAR
        BNZ     R3,LOOP
        ADDD        DAR,F2,DAR
```

O código acima produz e armazena um resultado a cada 4 ciclos.

2.3– Controle de malha

O código acima é redundante, no sentido que o valor corresponde ao número de elementos é decrementado duas vezes: uma automaticamente pelo hardware, ao decrementar VLR, e a outra pelo software, ao decrementar R3. A eliminação das instruções de decremento e teste pode dobrar o desempenho, neste caso.

Este resultado pode ser conseguido pela inclusão de uma nova instrução de controle,

```
DOLOOP n
```

onde n é a distância, medida em instruções, até a última instrução da malha. Esta instrução implica na execução repetida das n instruções que a seguem, até que o valor de VLR seja 0. Através de um recurso simples de hardware, que é manter na unidade de controle do processador uma cópia já decodificada da primeira instrução que segue ao DOLOOP, é possível fazer o desvio com latência zero.

Com esta instrução, a malha acima se torna

```
LD      F0,a
DOLOOP  #2
MULTD   F2,F0,DAR
ADDD    DAR,F2,DAR
```

Como no caso anterior, a latência de MULTD pode ser oculta por escalonamento ou por pipeline em software. Uma vez realizado, torna-se possível obter e armazenar um resultado a cada dois ciclos, com o desempenho limitado pela restrição de um único acesso de leitura à memória em cada ciclo.

Como comparação, o DLXV, um processador vetorial com características semelhantes, usando encadeamento e não considerando a latência inicial dos acessos à memória, produz e armazena um resultado a aproximadamente 4 ciclos.

2.4– Suporte em hardware para execução de malhas

Apesar de pipeline em software ser suficiente para ocultar a latência das operações de ponto flutuante, é muito mais simples dotar o processador de suporte em hardware para a execução simultânea de várias iterações de uma mesma malha. Esta técnica é conhecida como suporte policíclico [Deh89, Rau89], e é bastante simples de ser

implementada em um conjunto de registradores de ponto flutuante dedicados, chamados de registradores dinâmicos. Como as idéias estão descritas nas referências acima, esta seção apresentará apenas uma descrição sumária, numa variante adaptada para a arquitetura ortovetorial.

Antes de iniciar uma operação ortovetorial, os registradores dinâmicos são divididos em quadros; o número de elementos por quadro é o número máximo de registradores temporários exigidos simultaneamente para realizar todas as operações compreendidas pela malha interna em consideração. Este número é determinado pelo compilador, e uma instrução especial informa seu valor ao hardware. O quadro atual é indicado por um apontador que se move circularmente nos registradores dinâmicos; todas as referências a valores contidos em quadros são baseadas no valor corrente desse apontador, e seu valor é incrementado sempre que o valor de VLR é decrementado, indicando uma mudança de iteração.

Cada quadro contém os valores intermediários correspondentes a uma iteração; o quadro corrente contém os valores correspondentes à iteração corrente, e os quadros mais velhos contêm os valores correspondentes às iterações mais velhas.

Quando uma operação é iniciada e sua latência causa a introdução de uma espera de n ciclos, novas iterações vão sendo iniciadas e a iteração inicial continua após a espera de n ciclos, utilizando os valores presentes no n-étimo quadro mais antigo.

Como referências a quadros antigos usam números negativos para indicar o respectivo registrador, o processador pode determinar a idade de uma particular referência. Este fato permite eliminar a necessidade de predicados explícitos, como os usados no sistema Cydra 5 descrito nas referências acima; a idade da iteração é utilizada como um predicado implícito.

Com esses predicados implícitos, o hardware pode executar automaticamente prólogos e epílogos, que correspondem grosseiramente aos códigos de inicialização e de finalização necessários para pipeline por software, com a vantagem que funcionam uniformemente para qualquer número de iterações, inclusive 0 ou 1.

Quando se inicia a execução de uma malha interna somente são executadas as operações correspondentes ao quadro corrente; todas as operações que se referem a quadros mais velhos são ignoradas. Nas iterações subseqüentes são executadas somente as operações que correspondem ao quadro atual e aos quadros já calculados. Esta fase é chamada de prólogo, e assegura que somente as operações realmente necessárias sejam efetuadas. Uma vez atingida a situação de equilíbrio, todas as operações são efetuadas para todos os quadros. O número de quadros necessário, NF, pode ser calculado pelo compilador, e especificado ao hardware como um parâmetro adicional na instrução DOLOOP, ou pode ser determinado diretamente pelo hardware: é o número da iteração em que todas as instruções tenham sido executadas. Depois da malha ser executada o número de vezes especificado por VLR, entra-se na fase chamada de epílogo, que executa mais NF iterações, e complementa as iterações que ainda tenham operações pendentes. O hardware opera de uma maneira complementar: na i-ésima iteração do epílogo somente são executadas as operações que tenham idade i ou maior.

Embora este mecanismo seja aparentemente complexo, pode ser implementado com um simples registrador de deslocamento com alguma lógica adicional, sem risco de comprometimento do tempo de ciclo.

Este recurso evita que o compilador tenha de gerar códigos complexos para inicialização e finalização de malhas, e dispensa a necessidade de código particular quando a malha tem de ser executada um número pequeno de vezes. Dispensa ainda o escalonamento de instruções, uma vez que a latência de uma instrução não causa uma paralisação no funcionamento do processador.

Nos próximos exemplos será sempre suposto que este tipo de suporte de hardware esteja disponível, ignorando-se qualquer tipo de código de inicialização e de finalização, bem como problemas de escalonamento.

3 Tópicos Avançados

Esta seção examina vários aspectos ligados à implementações realísticas, bem como exemplos de códigos não vetorizáveis que são facilmente ortovetorizáveis.

3.1– Memórias reais

O primeiro problema é que embora seja possível construir memórias capazes de fornecer um elemento por ciclo, isto é conseguido com grandes latências. Caches são de pouca utilidade, ou até mesmo prejudiciais para este tipo de processamento, como indicam vários estudos independentes [Abu86, Die88, Rau89].

Como a arquitetura ortovetorial é desacoplada, a unidade de endereços pode enviar continuamente pedidos de dados à memória. Assim, passado uma demora inicial, é possível obter um valor por ciclo, com a latência de memória oculta pela própria natureza do projeto, desde que o subsistema de memória tenha sido projetado para suportar o fluxo de dados necessário.

A única exigência é que a fila de entrada seja maior do que a latência média de memória, expressa em ciclos de processador. Isto exige filas de entrada com 16 a 32 elementos, o que é facilmente factível.

3.2– Um exemplo com recorrências

Considere-se o núcleo 5 dos LLKs, que é não vetorizável devido a uma recorrência:

```
    Do 5 i = 2, n
5   X(i) = Z(i)*(Y(i)-X(i-1))
```

O código de inicialização para a unidade de endereços de leitura será

```
LDI     R3,#número de elementos
LDI     R1,#endereço de Y+stride de Y
LDI     R2,#stride de Y
MOV2LA  R1,R2
LDI     R1,#endereço de Z+stride de Z
LDI     R2,#stride de Z
MOV2LA  R1,R2
```

Para a unidade de endereços de escrita,

```
LDI     R1,#endereço de X+stride de X
LDI     R2,#stride de X
MOV2SA  R1,R2
```

Para VLR,

```
MOVI2S  VLR,R3
```

O código para a malha será

```
LD      F0,X(1)
DOLOOP  #3
SUBD    F2,DAR,F0
MULTD   F0,DAR,F2
SD      DAR,F0
```

Obtém-se um resultado a cada 3 ciclos; como se pode ver, malhas contendo recorrências podem ser facilmente ortovetorizáveis.

3.3– Exemplo com condicionais

O exemplo será um fragmento do núcleo 22. Como a inicialização é trivial não será apresentada. O fragmento é

```
IF ( U(K) .LT. EXPMAX * V(K))
            THEN Y(k) = U(k) / V(k)
            ELSE Y(k) = EXPMAX
......
```

O código é

```
        LD      F0, EXPMAX
        DOLOOP  #9
        MOVD    F2,DAR  ; preserva V(k)
        MULTD   F4,F0,F2
        MOVD    F6,DAR  ; preserva U(k)
        LTD     F6,F4
        BFPT    less
        MOVD    DAR,F0
        J       ahead
less    DIVD    DAR, F6, F2
ahead   ......
```

Supôs-se que apenas uma instrução seguisse ao IF para determinar o parâmetro de DOLOOP.

3.4– Operações de Gather e Scatter

As operações de gather e scatter podem ser facilmente realizadas pela combinação de operações ortovetoriais com operações comuns de load/store.

Como exemplo, considere-se uma malha formada apenas pelo comando

```
A(I) = B(K(I))
```

Que será compilado como (ignorando-se novamente as inicializações)

```
        DOLOOP  #3, M
        MVFP2I  R1, DAR
        LD      F0,B_offset(R1)
        MOVF    DAR,F0(idade M)
```

Supondo-se uma latência de M ciclos, esta malha pode fornecer um resultado por ciclo, após uma latência inicial, o que é um ótimo resultado para este tipo de operação.

4 Programando os LLKs

Para permitir uma melhor avaliação da arquitetura ortovetorial, esta seção mostra o resultado obtido para todos os núcleos LLK. Embora a codificacão tenha sido realizada manualmente, foi tomado todo o cuidado para não realizar nenhuma otimização além daquelas realizadas rotineiramente pelos compiladores de qualidade.

Para manter um nível realístico, os parâmetros para a unidade aritmética foram baseados num componente existente, o processador de ponto flutuante BIT B3130 [Bit89]: tempo de ciclo de 10 ns, latência 2 para operações aditivas e para multiplicações, 15 e 45 para divisões e raízes quadradas, respectivamente.

A arquitetura específica está descrita em [Cam92a, Cam92b], mas os resultados devem se manter próximos para qualquer implementação razoável de uma arquitetura ortovetorial, inclusive a baseada no DLXV apresentada acima.

4.1– Caso de um processador simples

Na tabela 1, NQ (Número de quadros) e TQ (Tamanho do quadro) mostram o número de quadros e respectivo tamanho, necessário para processar cada núcleo; seu produto é o número de registradores dinâmicos necessários para um processador com suporte policíclico. MFlops é desempenho assintótico em Megaflops; este número não resulta diretamente das colunas anteriores, uma vez que este resultado leva em consideração conflitos de acesso à memória. A coluna Cray-X foi incluída para fins de comparação.

Como era de se esperar, o desempenho do Cray é melhor nas malhas fortemente vetorizáveis, e muito pior nas malhas que são ortovetorizáveis mas não vetorizáveis. A superioridade do Cray não é devida a uma superioridade arquitetônica, mas ao fato de trabalhar com um ciclo muito menor.

Esta tabela permite visualizar algumas exigências de hardware para o projeto de um processador ortovetorial com suporte policíclico. O núcleo mais complexo, que é o 13, exige 42 registradores dinâmicos, o que sugere que o número 64 deve atender à quase totalidade dos casos. Nenhum dos núcleos exigiu mais de 16 pares de registradores de base e de stride na unidade de endereçamento.

Estes resultados podem ser facilmente estendidos para qualquer tipo de processamento numericamente intensivo? Não existe uma resposta concreta, mas [Ber91] sugere que existe uma forte correlação entre o comportamento de programas em processadores vetoriais e o comportamento de alguns núcleos LLK. Tudo indica que o mesmo pode se aplicar a processadores com arquitetura ortovetorial.

4.2– Caso superescalar

A extensão da arquitetura para um modelo superescalar é direta. O mesmo estudo anterior foi realizado para uma máquina superescalar de grau 2 (como em outras arquiteturas, há uma melhora significativa de desempenho ao se passar para o grau 2, mas muito pequena para graus maiores). Para manter uma arquitetura equilibrada em termos de capacidade de processamento e de acesso à memória, foi incluída uma segunda unidade de endereços para leitura, e supõe-se a memória capaz de fornecer, após uma certa latência, dois operandos e a receber uma palavra para escrita a cada ciclo.

A tabela 2 mostra os resultados.

Neste caso, o desempenho é muito próximo do Cray mesmo nos núcleos vetorizáveis. O número de registradores dinâmicos necessários permanece praticamente o mesmo.

Núcleo	Operações por resultado	Ciclos por resultado	NQ	TQ	MFlops	CRAY-X	Desempenho relativo
1	5	6	3	6	83	177	0.47
2	4	6	2	3	66	49	1.3
3	2	4	1	2	50	150	0.33
4	2	4	1	2	50	71	0.70
5	2	4	1	2	50	16	3.1
6	2	3	2	3	66	16	4.1
7	16	18	3	3	88	194	0.45
8	36	47	11	3	76	149	0.51
9	17	18	6	2	94	168	0.56
10	9	22	3	3	40	62	0.65
11	1	2	1	2	50	13	3.8
12	1	2	1	2	50	79	0.63
13	15	31	7	6	48	6	8.0
15	36	177	2	2	20	5	4.0
16	36	123	0	0	29	7	4.1
17	15	31	3	1	48	12	4.0
18	50	88	15	2	56	138	0.4
19	6	18	1	2	33	17	2.0
20	26	130	2	1	20	15	1.3
21	2	2	1	3	97	103	0.94
23	11	21	4	3	52	14	3.7
24	1	6	1	2	16	3	5.3

Tabela 1 - Desempenho da arquitetura ortovetorial

5 Conclusões

A arquitetura ortovetorial pode ser implementada com aproximadamente e mesmo custo que uma arquitetura vetorial, apresenta aproximadamente o mesmo desempenho no caso de malhas vetorizáveis, e um desempenho muito superior nos casos, muito freqüentes, em que uma malha é ortovetorizável mas não vetorizável.

Núcleo	Operações por resultado	Ciclos por resultado	TQ	NQ	MFlops	CRAY-X	Desempenho relativo
1	5	3	3	9	166	177	0.93
2	4	3	4	5	133	49	2.7
3	2	1	1	2	187	150	1.2
4	2	1	1	2	187	71	2.6
5	2	4	0	0	50	16	3.1
6	2	1	1	2	187	16	11.7
7	16	9	4	8	177	194	.91
8	36	20	9	4	180	149	1.2
9	17	9	8	3	185	168	1.1
10	9	10	3	5	84	62	1.3
11	1	2	0	0	50	13	3.8
12	1	1	2	6	93	79	1.2
13	15	11	4	10	132	6	22
15	36	113	2	2	31	5	6.2
16	36	120	0	0	30	7	4.3
17	15	23	1	2	65	12	5.4
18	50	44	7	5	113	138	0.81
19	6	18	1	2	33	17	1.9
20	26	130	2	1	20	15	1.3
21	2	1	1	2	172	103	1.7
23	11	17	2	3	64	14	4.6
24	1	5	1	2	20	3	6.7

Tabela 2 - Desempenho para implementação superescalar

Apesar de não ser essencial, a inclusão de suporte policíclico reduz consideravelmente a complexidade dos compiladores, tem um impacto muito pequeno sobre a complexidade do hardware e não compromete em nada o ciclo de máquina.

Como em outros estudos, pode-se concluir que há muito benefício em utilizar uma arquitetura superescalar com grau 2, e muito pouco benefício para graus mais elevados.

6 Bibliografia

Abu86 Abu-Sufah, W and Mahoney, A. D., "Vector Processing on the Alliant FX/8 Processor", Proc. Int'l Conf. Parallel Processing, 559-563, 1986.

Ber91 Berry, M. W. (ed.) "Scientific Workload Characterization By Loop-Based Analyses. Technical Report CS-91-140, Computer Science Department, Univ. of Tennessee.

Bit89 Bipolar Integrated Technology, Inc. "B2130/B3130/B4130 Single Chip Floating Point Processors", 1989.

Cam92a Campos, G. L. Asynchronous Polycyclic Architecture: an Overview. Information Processing 92, J. van Leewen (ed), Vol 1, 518-524, Madrid, set 1992.

Cam92b Campos, G. L. Asynchronous Polycyclic Architecture. Parallel Processing: CONPAR 92 - VAPP V (Lecture Notes in Computer Science, vol 634), Springer-Verlag, set 1992.

Die88 Diede, T., et al. "The Titan Graphics Supercomputer Architecture", IEEE Computer 21 (9):13-30, September 1988.

Lam88 Lam, M. "Software Pipelining: An Effective Scheduling Technique for VLIW Machines", Sigplan Conf. on Programming Language Design and Implementation, 318-328, June 89.

Mah86 McMahon, F. H. "The Livermore FORTRAN Kernels: A Computer Test of the Numerical Performance Range," Lawrence Livermore Nat'l Laboratory Report No. UCRL-53745, Livermore, CA, Dec. 1986.

Deh89 Dehnert, J. C., Hsu, P. Y. T., Bratt, J. P., "Overlapped Loop Support in the Cydra 5", 3rd Int. Conf. on Architectural Support for Programming Languages and Operating Systems, 26-38, April 1989.

Fly66 Flynn, M. J. "Very High-speed Computing Systems", Proc. IEEE 54 (12) 1901-09, December 1966.

Hen90 Hennessy, J. L. and Patterson, D. A. "Computer Architecture: A Quantitative Approach". Morgan, Kaufmann Publishers, California, 1990.

Rau89 Rau, B. R., Yen, D. W. L. and Towle, R. A. "The Cydra 5 Departmental Supercomputer", IEEE Computer 22 (1):12-35, February 1989.

Smi84 Smith, J. E., "Decoupled Access/Execute Architecture Computer Architectures", ACM Trans. Computer Systems, 2(4):298-308, Nov 1984.

Wul88 Wulf, Wm. A., "The WM Computer Architecture", Computer Architecture News, 16(1):70:84, March 1988.