A Story of Parallelism: from Imperative and Functional Languages

Cosmin E. Oancea `cosmin.oancea@diku.dk`

HIPERFIT, Department of Computer Science
University of Copenhagen

04.02.13 – Invited Lecture for the Advanced Compiler Course

Motivation

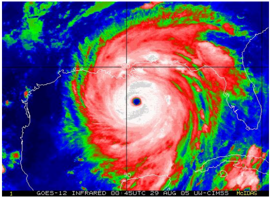
Parallel hardware is here to stay, e.g., general-purpose graphic processing units (GPGPU).

Powerful graphical cards are fundamental to gaming ...

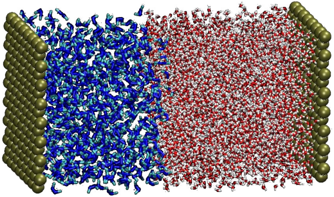
Motivation

... and (GPGPUs) have also been used in other areas of less impact:

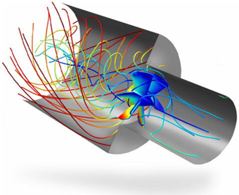
weather prediction,



bioinformatics,



fluid-dynamic simulations,



finance

1. Brief History of Hardware and High-Level Comparison
2. Imperative Context: Direction-Vector Analysis of Parallelism
3. Imperative Context: Summarization of Array Indexes
4. Functional Context: List Homomorphisms and Map-Reduce Style

Brief Hardware History

- How did general-purpose graphics-processing units (GPGPU) come to being?

- In the beginning was the Single-CPU,
  and the single-CPU was sequentially-programmable,
  and sequential code was programming.

- Then the frequency of the single-CPU could not be further increased and Multi-cores/processors became mainstream.

- And "Hardware Engineers" worked hard to support the illusion that random-access to memory has uniform cost:
  - thrown many transistors to memory-hierarchy coherency and such
  - ... and ultimately (arguably) they have failed (to scale up)!

- But, but, but ... is the sequentially-written code going to benefit?

Multicore Issues

What is the main cost of manual parallelization?

- learning various tools, e.g., OpenMP, MPI, Intel-Basic Blocks?
- rewriting the code using those tools?
- ensuring that the parallel and sequential programs are equivalent?

Multicore Issues

What is the main cost of manual parallelization?

- learning various tools, e.g., OpenMP, MPI, Intel-Basic Blocks?
- rewriting the code using those tools?
- ensuring that the parallel and sequential programs are equivalent?

Last one! Even reasoning about multicore hardware is nontrivial and error-prone. Furthermore, hardware instructions, such as memory fences and compare-and-swap (CAS) do not seem to scale.

Memory Sequential Consistency (Memory Reordering)

```
          //Initially x = 0; y = 0;
//CORE 0                            //CORE 1
x = 1;                              y = 1;
//mfence;                           //mfence;
write(y);                           write(x);
```

Multicore Issues

What is the main cost of manual parallelization?

- learning various tools, e.g., OpenMP, MPI, Intel-Basic Blocks?
- rewriting the code using those tools?
- ensuring that the parallel and sequential programs are equivalent?

Last one! Even reasoning about multicore hardware is nontrivial and error-prone. Furthermore, hardware instructions, such as memory fences and compare-and-swap (CAS) do not seem to scale.

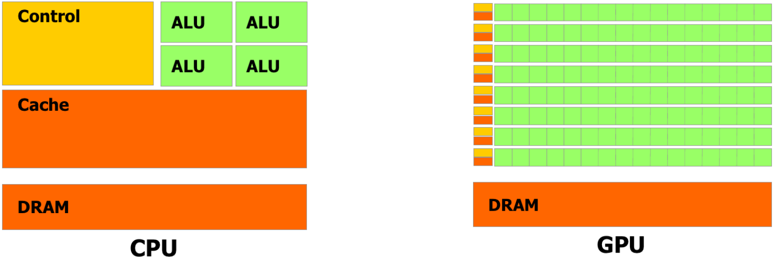
Memory Sequential Consistency (Memory Reordering)

```
        //Initially x = 0; y = 0;
//CORE 0                           //CORE 1
x = 1;                             y = 1;
//mfence;                          //mfence;
write(y);                          write(x);
```

No possible interleaving of instructions can result in both `x` and `y` being 0 at the end. And still it happens. Fixed with `mfences`.

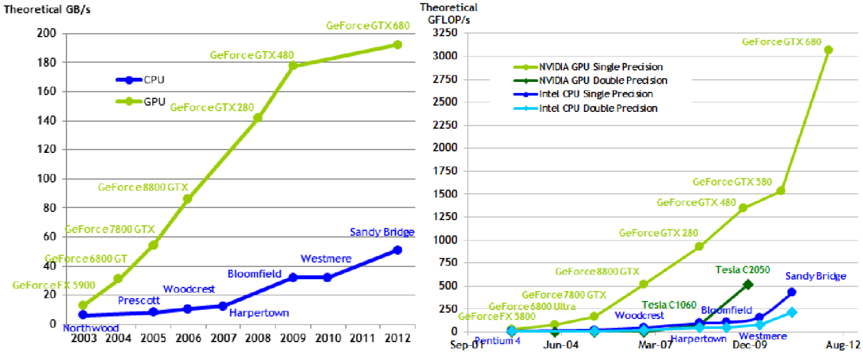
Brief Hardware History (Continuation)

- Then "Hardware Engineers" fixed scalability with GPGPU but took away programming convenience:
  - single-instruction multiple-data (SIMD):
    - cannot make an omelet with one core and a steak with the next,
    - but you can make first an omelet, then a steak, both very fast.
  - non-uniform, explicitly programmable memory hierarchy.
  - no dynamic allocation, no stack.
  - can only synchronize "locally" via barriers,
  - # cores $\sim$ thousands $\Rightarrow$ not enough to $\|$ only the outermost loop!



Theoretical Hardware Comparison: CPU vs GPGPU

GPGPU shows superior peak bandwidth and compute power vs. CPU.



However, peak bandwidth requires coalesced accesses, i.e., if the cores executing the (same) instruction access consecutive memory locations then the data is brought in via *a single* memory transfer.

Example of GPU programming

gpGPU no dynamic allocation: what to do with local array variables?

Example of GPU programming

gpGPU no dynamic allocation: what to do with local array variables?

Loop Fusion	Loop Distribution

Loop Fusion

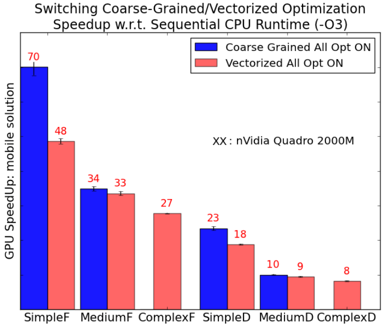
```
DO i = 1, N   // Parallel
  FLOAT A[M];  // local
  DO j = 1, M
    A[j] = ... f(i) ...
  ENDDO

  DO j = 1, M
    A[j] = ... g(A[j]) ...
  ENDDO

  FLOAT sum = 0.0
  DO j = 1, M
    sum += A[j];
  ENDDO
  X[i] = sum;
ENDDO
```

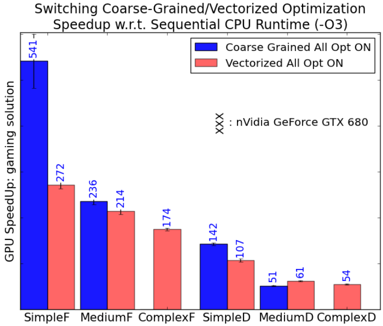
Loop Distribution

```
FLOAT A[N,M];  // global
DO i = 1, N   // Parallel
  DO j = 1, M
    A[i,j] = ... f(i) ...
  ENDDO
ENDDO
DO i = 1, N   // Parallel
  DO j = 1, M
    A[i,j] = ... g(A[i,j]) ...
  ENDDO
  real sum = 0.0
  DO j = 1, M
    sum += A[i,j];
  ENDDO
  X[i] = sum;
ENDDO
```

Impact of Loop Fusion/Distribution

Speedup on mobile and gaming GPGPUs with loop fusion ON (blue) and loop distribution ON (red). HIGHER is BETTER!

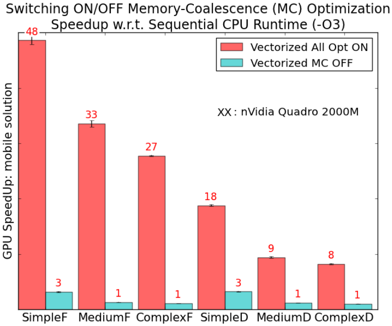


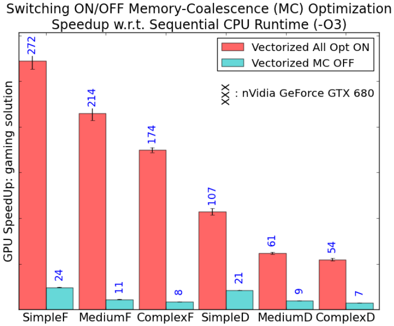


Impact is input-sensitive; but a cost model can chose at runtime the better alternative.

Impact of Memory-Coalescing Optimization

Speedup on mobile and gaming GPGPUs with memory-coalescing optimization ON (red) and OFF (blue). HIGHER is BETTER!





Not natural but effective transformation, hence suited to be implemented in the repertoire of an optimizing compiler.

Problem Statement

Iterations are ordered *lexicographically*, w.r.t. how they occur in the sequential execution, e.g., first loop next, `(j=1,i=4) < (j=2,i=3)`.

Three Loop Examples

```
DO i = 1, N
  DO j = 1, N
    A[j,i] = A[j,i] ..
  ENDDO
ENDDO
```

```
DO i = 2, N
  DO j = 2, N
    A[j,i] = A[j-1,i-1]...
    B[j,i] = B[j-1,i]...
ENDDO ENDDO
```

```
DO i = 2, N
  DO j = 1, N
      A[i,j] = A[i-1,j+1]...
  ENDDO
ENDDO
```

- Which of the three loop nests is amenable to parallelization?
- Loop interchange is one of the most simple and useful code transformations, e.g., used to enhance locality of reference, parallel-loop granularity, and even to "create" parallelism.
- In which loop nest is it safe to interchange the loops?

Loop-Nest Dependencies

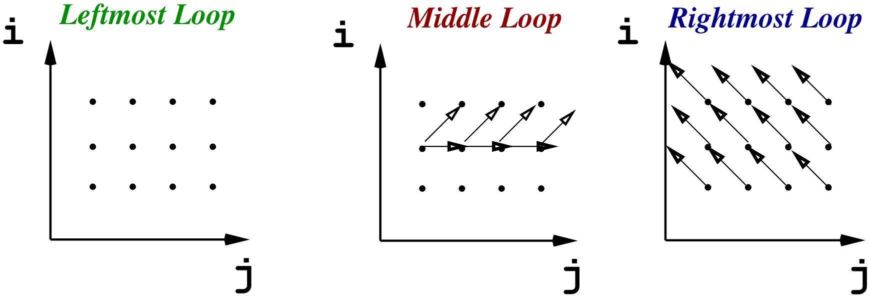
Iterations are ordered *lexicographically*, w.r.t. how they occur in the sequential execution, e.g., first loop next, `(j=1,i=4) < (j=2,i=3)`.

Three Loop Examples

```
DO i = 1, N
  DO j = 1, N
    A[j,i] = A[j,i] ..
  ENDDO
ENDDO
```

```
DO i = 2, N
  DO j = 2, N
    A[j,i] = A[j-1,i-1]...
    B[j,i] = B[j-1,i]...
ENDDO ENDDO
```

```
DO i = 2, N
  DO j = 1, N
      A[i,j] = A[i-1,j+1]...
  ENDDO
ENDDO
```



Definition of a Dependency

Load-Store Classification of Dependencies

```
True Dependency (RAW)      Anti Dependency (WAR)      Output dependency (WAW)
S1     X  = ..             S1     .. = X              S1     X = ...
S2     .. = X              S2     X  = ..             S2     X = ...
```

Th. Loop Dependence: There is a dependence from statement $S1$ to $S2$ in a loop nest *iff* $\exists$ iterations i, j such that:

1. $i < j$ or $i = j$ and $\exists$ a path from $S1$ to $S2$ such that
2. $S1$ accesses memory location M on iteration i, and
3. $S2$ accesses memory location M on iteration j, and
4. one of these accesses is a write.

We are most interested in cross iteration dependencies, i.e., $i < j$.

If $i = j$ the dependency is within the same iteration, i.e., does not affect parallelism, since an iteration is executed sequentially.

Direction Vectors

Three Loop Examples

```
  DO i = 1, N              DO i = 2, N                DO i = 2, N
    DO j = 1, N              DO j = 2, N                DO j = 1, N
S1     A[j,i]=A[j,i]..  S1    A[j,i]=A[j-1,i]...   S1     A[i,j]=A[i-1,j+1]...
    ENDDO               S2    B[j,i]=B[j-1,i-1]...      ENDDO
  ENDDO                    ENDDO ENDDO                ENDDO
```

Dependencies depicted via an edge *from* the stmt that executes first in the loop nest, i.e., *the source*, *to* the one that executes later, *the sink*.

Def. Dependence Direction: Assume $\exists$ a dependence from $S1$ in iteration q to $S2$ in t $(q < t)$. *Dependence-direction vector* $D(q,t)$:

1. $D(q,t)_k =$ "<" if $t_k - q_k > 0$,
2. $D(q,t)_k =$ "=" if $t_k - q_k = 0$,
3. $D(q,t)_k =$ ">" if $t_k - q_k < 0$.

Let's write the direction vectors for the three loop nests.

Parallelism and Loop Interchange

Direction Vectors/Matrix for Three Loops

```
  DO i = 1, N
    DO j = 1, N
S1     A[j,i]=A[j,i]..
    ENDDO
  ENDDO

For S1→S1: j1 = j2
           i1 = i2
(i2,j2)-(i1,j1)=
[=,=]
```

```
  DO i = 2, N
    DO j = 2, N
S1    A[j,i]=A[j-1,i]...
S2    B[j,i]=B[j-1,i-1]...
  ENDDO ENDDO

For S1→S1: j1 = j2-1
           i1 = i2
  (i2,j2)-(i1,j1)=[=,<]
For S2→S2: j1 = j2-1
           i1 = i2-1
  (i2,j2)-(i1,j1)=[<,<]
```

```
  DO i = 2, N
    DO j = 1, N
S1     A[i,j]=A[i-1,j+1]...
    ENDDO
  ENDDO

For S1→S1: i1 = i2-1
           j1 = j2+1
(i2,j2)-(i1,j1)=[<,>]
```

Th. Parallelism: A loop in a loop nest is parallel *iff* all its directions are either = or there exists an outer loop whose corresp. direction is <.

Th. Loop Interchange: A column permutation of the loops in a loop nest is legal *iff* permuting the direction matrix in the same way *does NOT result* in a > direction as the leftmost non-= direction in a row.

Parallelism and Loop Interchange

Direction Vectors/Matrix for Three Loops

```
  DO i = 1, N
    DO j = 1, N
S1    A[j,i]=A[j,i]..
    ENDDO
  ENDDO

For S1→S1: j1 = j2
           i1 = i2
(i2,j2)-(i1,j1)=
[=,=]
```

```
  DO i = 2, N
    DO j = 2, N
S1    A[j,i]=A[j-1,i]...
S2    B[j,i]=B[j-1,i-1]...
  ENDDO ENDDO

For S1→S1: j1 = j2-1
           i1 = i2
  (i2,j2)-(i1,j1)=[=,<]
For S2→S2: j1 = j2-1
           i1 = i2-1
  (i2,j2)-(i1,j1)=[<,<]
```

```
  DO i = 2, N
    DO j = 1, N
S1    A[i,j]=A[i-1,j+1]...
    ENDDO
  ENDDO

For S1→S1: i1 = i2-1
           j1 = j2+1
(i2,j2)-(i1,j1)=[<,>]
```

Interchange is safe for the first and second nests, but not for the third!

e.g., [=,<] → [<,=] (for the second loop nest)

[<,<] [<,<]

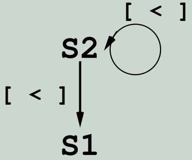
After interchange, loop *j* of the second loop nest is parallel.

Corollary: A parallel loop can be always interchanged inwards.

Dependency Graph and Loop Distribution

Vectorization Example

```
  DO i = 2, N
S1  A[i] = B[i-1] ...
S2  B[i] = B[i-1] ...
  ENDDO

For S2→S1: i1 = i2-1, [<]
For S2→S2: i1 = i2-1, [<]
```




```
  DO i = 2, N
S2  B[i] = B[i-1] ...
  ENDDO

  DOALL i = 2, N
S1  A[i] = B[i-1] ...
  ENDDOALL
```

Def. Dependency Graph: edges from the source of the dependency, i.e., early iteration, to the sink, i.e., later iteration.

Th. Loop Distribution: Statements that are in a dependence cycle remain in one (sequential) loop. The others are distributed to separate loops in graph order; if no cycle then parallel loops.

Corollary: It is always legal to distribute a parallel loop; potentially you need array expansion if output dependencies are present.

Block Tiling via Loop Distribution and Interchange

Matrix Multiplication Example: First Tile all Loops (Always Safe)

```
DO i = 1, N    // Parallel
  DO j = 1, N  // Parallel
    C[i,j] = 0.0

    DO k = 1, N
      C[i,j] += A[k,j] * B[i,k]
    ENDDO
  ENDDO
ENDDO
```

LOOP -------> TILING

```
DO ii = 1, N, L
  DO i = ii, ii+L-1
    DO jj = 1, N, L
      DO j = jj, jj+L-1
        C[i,j] = 0.0
        DO kk = 1, N, L
          DO k = kk, kk+L-1
            C[i,j] += A[k,j]*B[i,k]
ENDDO ENDDO ENDDO ENDDO ENDDO ENDDO
```

Loops i and j are parallel: Move Inside, Then Distribute

```
DO ii = 1, N, L
  DO jj = 1, N, L
    DO i = ii, ii+L-1
      DO j = jj, jj+L-1
        C[i,j] = 0.0
        DO kk = 1, N, L
          DO k = kk, kk+L-1
            C[i,j] += A[k,j]*B[i,k]
ENDDO ENDDO ENDDO ENDDO ENDDO ENDDO
```

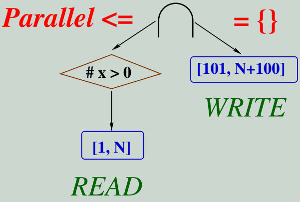
DISTRIBUTE LOOPS i,j ---------> AND MOVE THEM INSIDE kk

```
DOALL ii = 1, N, L
  DOALL jj = 1, N, L
    DO i = ii, ii+L-1
      DO j = jj, jj+L-1
        C[i,j] = 0.0
    ENDDO ENDDO
    DO kk = 1, N, L
      DO i = ii, ii+L-1
        DO j = jj, jj+L-1
          DO k = kk, kk+L-1
            C[i,j] += A[k,j]*B[i,k]
ENDDO ENDDO ENDDO ENDDO ENDDO ENDDO
```

Interprocedural Summarization of Array Indexes

Independence-Summary Simple Example

```
DO i = 1, N
  A(i+100) = ...
  IF (x > 0) THEN
    ... = A(i)
  ENDIF
ENDDO
```



- Techniques that analyze read-write pairs of accesses become very conservative on larger loops with non-trivial control flow.
- *Alternative:* inter-procedural summarization + model loop independence via an equation on summaries of shape $S = \emptyset$
- Decrease overhead by extracting lightweight predicates that prove independence at runtime, e.g., $\texttt{x} \leq \texttt{0} \quad \vee \quad \texttt{N} < \texttt{100}$.

Show Calculix from SPEC2006!

Building RO, RW, WF Summaries Interprocedurally

Summaries (RO, RW, WF) are

- constructed via a bottom-up parse of the CALL and CD graphs,
- structural data-flow equations dictate how to compose consecutive regions, aggregate/translate across loops/callsites, ...

Building RO, RW, WF Summaries Interprocedurally

Summaries (RO, RW, WF) are

- constructed via a bottom-up parse of the CALL and CD graphs,
- structural data-flow equations dictate how to compose consecutive regions, aggregate/translate across loops/callsites, ...

Simplified solvh_do20 from dyfesm

```
DO i = 1, N
    CALL geteu (XE(IA(i)), NP, SYM)
    CALL matmul(XE(IA(i)), NS)
ENDDO

SUBROUTINE matmul(XE, NS)
  INTEGER NS, XE(*)
  DO j = 1, NS
    ...   = XE(j) ...
    XE(j) = ...
  ENDDO
END
```

```
SUBROUTINE geteu(XE, NP, SYM)
  INTEGER NP, SYM, XE(16, *)

  IF (SYM .NE. 1) THEN
    DO i = 1, NP
      DO j = 1, 16
        XE(j, i) = ...
      ENDDO
    ENDDO
  ENDIF
END
```

Summarizing Subroutine geteu

WF summary for geteu; $RO_{geteu} = RW_{geteu} = \emptyset$

S_{geteu}	`SUBROUTINE geteu(XE, NP, SYM)`
	`  INTEGER NP, SYM, XE(16, *)`
S_{IF}	`  IF (SYM .NE. 1) THEN`
S_{Li}	`    DO i = 1, NP`
S_{Lj}	`      DO j = 1, 16`
S_{WF}	`        XE(j, i) = ...`
	`      ENDDO`
	`    ENDDO`
	`  ENDIF`
	`END`

$$WF^{XE}_{S_{WF}} = \{16 * i + j - 1\}$$

- Loop Aggregation uses (intuitively) interval arithmetic:
- Loop i: $\{16 * i + j - 1 \;\mid j \in \{1..\,16\}\} \rightarrow 16 * i + [0, 15]$
- Loop j: $\{16 * i + [0, 15] \mid i \in \{1..NP\}\} \rightarrow [0, 16 * NP - 1]$
- Branches introduce predicated nodes, e.g., $WF^{XE}_{S_{if}} = WF^{XE}_{S_{geteu}}$

Summarizing Subroutine geteu

WF summary for geteu; $RO_{geteu} = RW_{geteu} = \emptyset$

S_{geteu}, S_{IF}, S_{Li}, S_{Lj}, S_{WF}:

```
S_geteu  SUBROUTINE geteu(XE, NP, SYM)
           INTEGER NP, SYM, XE(16, *)
S_IF        IF (SYM .NE. 1) THEN
S_Li          DO i = 1, NP
S_Lj            DO j = 1, 16
S_WF               XE(j, i) = ...
                ENDDO
              ENDDO
           ENDIF
         END
```

$$WF^{XE}_{S_{Lj}} = 16 * i + [0, 15]$$

$$WF^{XE}_{S_{WF}} = \{16 * i + j - 1\}$$

- Loop Aggregation uses (intuitively) interval arithmetic:
- Loop i: $\{16 * i + j - 1 \ \mid j \in \{1..\ 16\}\} \rightarrow 16 * i + [0, 15]$
- Loop j: $\{16 * i + [0, 15] \mid i \in \{1..NP\}\} \rightarrow [0, 16 * NP - 1]$
- Branches introduce predicated nodes, e.g., $WF^{XE}_{S_{if}} = WF^{XE}_{S_{geteu}}$

Summarizing Subroutine `geteu`

WF summary for `geteu`; $RO_{geteu} = RW_{geteu} = \emptyset$

```
S_geteu  SUBROUTINE geteu(XE, NP, SYM)
           INTEGER NP, SYM, XE(16, *)
S_IF        IF (SYM .NE. 1) THEN
S_Li          DO i = 1, NP
S_Lj            DO j = 1, 16
S_WF                XE(j, i) = ...
                ENDDO
              ENDDO
           ENDIF
         END
```

$$WF^{XE}_{S_{Li}} = [0, 16 * NP - 1]$$

$$WF^{XE}_{S_{Lj}} = 16 * i + [0, 15]$$

$$WF^{XE}_{S_{WF}} = \{16 * i + j - 1\}$$

- Loop Aggregation uses (intuitively) interval arithmetic:
- Loop i: $\{16 * i + j - 1 \mid j \in \{1..16\}\} \rightarrow 16 * i + [0, 15]$
- Loop j: $\{16 * i + [0, 15] \mid i \in \{1..NP\}\} \rightarrow [0, 16 * NP - 1]$
- Branches introduce predicated nodes, e.g., $WF^{XE}_{S_{if}} = WF^{XE}_{S_{geteu}}$

Summarizing Subroutine `geteu`

WF summary for geteu; $RO_{geteu} = RW_{geteu} = \emptyset$

S_{geteu}	`SUBROUTINE geteu(XE, NP, SYM)`
	`INTEGER NP, SYM, XE(16, *)`
S_{IF}	`IF (SYM .NE. 1) THEN`
S_{Li}	`DO i = 1, NP`
S_{Lj}	`DO j = 1, 16`
S_{WF}	`XE(j, i) = ...`
	`ENDDO`
	`ENDDO`
	`ENDIF`
	`END`

$$WF^{XE}_{S_{IF}} = \begin{array}{c}(SYM \neq 1)\\ \downarrow \\ [0, 16*NP-1]\end{array}$$

$$WF^{XE}_{S_{Li}} = [0, 16*NP-1]$$

$$WF^{XE}_{S_{Lj}} = 16*i + [0,15]$$

$$WF^{XE}_{S_{WF}} = \{16*i+j-1\}$$

- Loop Aggregation uses (intuitively) interval arithmetic:
- Loop i: $\{16*i+j-1 \mid j \in \{1..16\}\} \rightarrow 16*i+[0,15]$
- Loop j: $\{16*i+[0,15] \mid i \in \{1..NP\}\} \rightarrow [0, 16*NP-1]$
- Branches introduce predicated nodes, e.g., $WF^{XE}_{S_{if}} = WF^{XE}_{S_{geteu}}$

Summarizing Subroutine `matmult`

RW summary for `matmul`; $RO_{matmul} = WF_{matmul} = \emptyset$

S_{matmul}
S_{loop}
S_{RO}
S_{WF}

```
SUBROUTINE matmul(XE, NS)
  INTEGER NS, XE(*)
   DO j = 1, NS
      ...    = XE(j) ...
      XE(j) = ...
  ENDDO
END
```

$RO^{XE}_{S_{RO}} = \{j-1\}$

$WF^{XE}_{S_{WF}} = \{j-1\}$

- Composing read-only RO_{S_1} and write-first WF_{S_2} regions:
- $RO = RO_{S_1} - WF_{S_2}$, $WF = WF_{S_2} - RO_{S_1}$, $RW = RO_{S_1} \cap WF_{S_2}$
- In our case $RO = \emptyset$, $WF = \emptyset$, $RW = \{j-1\}$
- Over loop `DO j`: $RO_{loop} = \emptyset$, $WF_{loop} = \emptyset$, $RW_{loop} = [0, NS-1]$

Summarizing Subroutine `matmult`

RW summary for `matmul`; $RO_{matmul} = WF_{matmul} = \emptyset$

S_{matmul}	`SUBROUTINE matmul(XE, NS)`	
	`  INTEGER NS, XE(*)`	
S_{loop}	`   DO j = 1, NS`	$S_{RO} \diamond S_{WF} = \{\emptyset, \emptyset, RW = \{j-1\}\}$
S_{RO}	`      ...    = XE(j) ...`	
S_{WF}	`      XE(j) = ...`	$RO^{XE}_{S_{RO}} = \{j-1\}$
	`  ENDDO`	$WF^{XE}_{S_{WF}} = \{j-1\}$
	`END`	

- Composing read-only RO_{S_1} and write-first WF_{S_2} regions:
- $RO = RO_{S_1} - WF_{S_2}$, $WF = WF_{S_2} - RO_{S_1}$, $RW = RO_{S_1} \cap WF_{S_2}$
- In our case $RO = \emptyset$, $WF = \emptyset$, $RW = \{j-1\}$
- Over loop `DO j`: $RO_{loop} = \emptyset$, $WF_{loop} = \emptyset$, $RW_{loop} = [0, NS-1]$

Summarizing Subroutine `matmult`

RW summary for `matmul`; $RO_{matmul} = WF_{matmul} = \emptyset$

S_{matmul}	`SUBROUTINE matmul(XE, NS)`	$RW^{XE}_{S_{loop}} = [0, NS-1]$
	`INTEGER NS, XE(*)`	
S_{loop}	`DO j = 1, NS`	$S_{RO} \diamond S_{WF} = \{\emptyset,\ \emptyset,\ RW = \{j-1\}\}$
S_{RO}	`... = XE(j) ...`	$RO^{XE}_{S_{RO}} = \{j-1\}$
S_{WF}	`XE(j) = ...`	$WF^{XE}_{S_{WF}} = \{j-1\}$
	`ENDDO`	
	`END`	

- Composing read-only RO_{S_1} and write-first WF_{S_2} regions:
- $RO = RO_{S_1} - WF_{S_2}$, $WF = WF_{S_2} - RO_{S_1}$, $RW = RO_{S_1} \cap WF_{S_2}$
- In our case $RO = \emptyset$, $WF = \emptyset$, $RW = \{j-1\}$
- Over loop `DO j`: $RO_{loop} = \emptyset$, $WF_{loop} = \emptyset$, $RW_{loop} = [0, NS-1]$

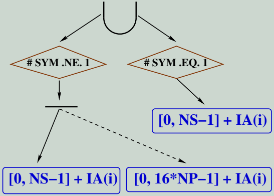
Summarizing Accesses for the Target Loop

RW summary for loop DO i: $RW^i = ?$

	`INTEGER NS, NP, IA(*), XE(*)`
S_{loop}	`DO i = 1, N`
S_{WF}	`  CALL geteu (XE(IA(i)),NP,SYM)`
S_{RW}	`  CALL matmul(XE(IA(i)),NS)`
	`ENDDO`

$S_{WF} \ \diamond \ S_{RW} \ = \ \{\emptyset, \ WF^i = WF_{geteu}, \ RW^i\}$

RW^i=



In our case, a sufficient condition for XE independence is:

$$\cup_{i=1}^{N}(RW_i \ \cap \ (\cup_{k=1}^{i-1} RW_k)) = \emptyset \quad \Leftrightarrow \quad RW_i = \emptyset \quad \Leftrightarrow$$

$$\Leftrightarrow \quad \mathtt{SYM} \neq 1 \ \wedge \mathtt{NS} \leq 16 * \mathtt{NP}$$

Summary-Based Independence Equations

Flow and Anti Independence Equation for loop of index i:

$$\begin{aligned} S_{find} = &\{(\cup_{i=1}^{N} WF_i) \ \cap \ (\cup_{i=1}^{N} RO_i)\} \cup \\ &\{(\cup_{i=1}^{N} WF_i) \ \cap \ (\cup_{i=1}^{N} RW_i)\} \cup \\ &\{(\cup_{i=1}^{N} RO_i) \ \cap \ (\cup_{i=1}^{N} RW_i)\} \cup \\ &\{\cup_{i=1}^{N} (RW_i \ \cap \ (\cup_{k=1}^{i-1} RW_k))\} \quad = \emptyset \end{aligned} \tag{1}$$

Output Independence Equation for loop of index i:

$$S_{oind} = \{\cup_{i=1}^{N} (WF_i \ \cap \ (\cup_{k=1}^{i-1} WF_k))\} \quad = \emptyset \tag{2}$$

Computing S_{find} and S_{oind} solves a more difficult problem than we need, i.e., computes the indexes involved in cross-iteration deps.

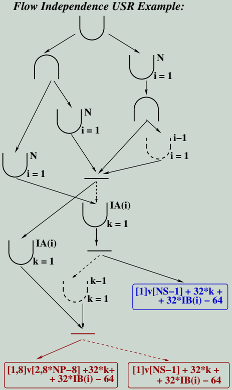
Loop Independence: when are S_{find} and S_{oind} are empty?

Key Idea: Predicate-Centric Approach

Approach centered on extracting arbitrarily-shaped predicates.

Key Idea

- *Source of inaccuracy:* summary representation not closed under composition w.r.t. set operations.
- *Language* representation for summaries ... precise but expensive to compute at runtime
- "Let's *reason* about it!"
  $8 * NP < NS + 6 \Rightarrow A - B = \emptyset \Rightarrow S = \emptyset!$



Show Calculix from SPEC2006!

1. Brief History of Hardware and High-Level Comparison
2. Imperative Context: Direction-Vector Analysis of Parallelism
3. Imperative Context: Summarization of Array Indexes
4. Functional Context: List Homomorphisms and Map-Reduce Style

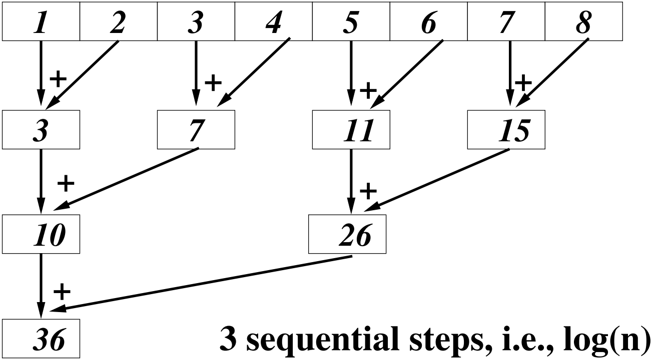
Bird-Meertens Formalism (BMF)

BMF: small collection of (i) second-order functions on lists, (ii) algebraic identities and theorems, and (iii) a concise notation.

BMF Notation:

`id`	identity function, i.e., $\texttt{id} : T \rightarrow T, \texttt{id}\ x = x$
.	backward functional composition: $(f\ .\ g)\ x = f\ (g\ x)$
$\oplus, \otimes, \odot$	binary associative operators, $\odot :: T \rightarrow T \rightarrow T$
$\texttt{zipWith}\ \odot$	application of $\odot$ to a pair of equal-length lists: $\texttt{zipWith}\ \odot\ [x_1, .., x_n]\ [y_1, .., y_n]\ =\ [x_1 \odot y_1, .., x_n \odot y_n]$.
$\texttt{map}\ f$	$f :: T_1 \rightarrow T_2$, $\texttt{map} :: [T_1] \rightarrow [T_2]$, $\texttt{map}\ f\ [x_1, .., x_n]\ =\ [(f\ x_1), .., (f\ x_n)]$
$\texttt{red}\ \odot$	reduce with binary associative operator $\odot$, $\texttt{red} :: (T \rightarrow T \rightarrow T) \rightarrow T \rightarrow [T] \rightarrow T$ $red\ \odot\ e_\odot\ [x_1, ..., x_n]\ =\ e_\odot\ \odot\ x_1\ \odot\ ...\ \odot\ x_n$

Reducing in Parallel



We build programs by combining map and reduce. For example,
scan (+) [1, 2, 3, 4] → [1, 1+2, 1+2+3, 1+2+3+4] → [1, 3, 6, 10]
can be efficiently implemented as a map reduce.

List-Homomorphism and Map-Reduce Equivalence

Those functions that promote through list concatenation (++):

Definition (List Homomorphism)

$h :: [T_1] \to T_2$ *over finite lists is a* list homomorphism *if there exists an associative binary operator* $\odot :: T_2 \to T_2 \to T_2$*, such that:*

$$h\ (x{+}{+}y) = (h\ x)\ \odot\ (h\ y)$$

We denote $h = hom\ (\odot)\ f\ e_\odot$*, where* $f\ x = h\ [x]$ *and* $e_\odot = h\ []$.

Theorem (1st List Homomorphism Theorem (LHTh1))

Any list homomorphism can be written as the composition of a reduction and a map: $h = hom\ (\odot)\ f\ e_\odot\ =\ (red\ (\odot)\ e_\odot)\ .\ (map\ f)$
Conversely, each such composition is a homomorphism.

Theorem tells how to parallelize LHs based on `map-reduce` skeletons.

List Homomorphisms (LH)

Examples of List Homomorphisms as Map-Reduce

```
len :: [T] -> Integer
len []      = 0
len [x]     = 1
len (x++y) = (len x) + (len y)
len ≡ (red (+) 0) . (map (fn x ⇒ 1))
-- logical and ∧ :: T -> T -> Bool
-- all elems satisfy p :: T → Bool ?
all_p :: [T] -> Bool
all_p []      = True
all_p [x]     = p x
all_p (x++y) = (all_p x) ∧ (all_p y)
all_p ≡ (red (∧) True) .
         (map (fn x ⇒ p x))
↑ :: Integer -> Integer -> Integer
x ↑ y = if (x > y) then x else y
maxList :: [Integer] -> Integer
maxList []      = −∞
maxList [x]     = x
maxList (x++y) = (maxList x) ↑
                 (maxList y)
maxList ≡ (red (↑) −∞) . (map id)
```

```
-- merges two sorted lists
-- into a sorted list
merge :: [T] -> [T] -> [T]
merge :: Ord a => [a] -> [a] -> [a]
merge [] y  = y
merge x  [] = x
merge (x::xs) (y::ys) =
  if ( x <= y )
  then x :: merge xs (y::ys)
  else y :: merge (x::xs) ys

-- [.] x = [x]
mSort :: [T] -> T
mSort []      = []
mSort [x]     = [x]
mSort (x++y) = merge (mSort x)
                     (mSort y)

mSort ≡ (red merge []) . (map [.])
```

List Homomorphism Invariants

Theorem (Map Fusion/Distribution)

Given unary functions f and g then:

$$(\mathtt{map}\ f)\ .\ (\mathtt{map}\ g)\ \equiv\ \mathtt{map}\ (f\ .\ g)$$

Theorem (List-Homomorphism Promotions)

Given unary function f and an associative binary operator $\odot$ then:

$$(\mathtt{map}\ f)\ .\ (\mathtt{red}\ (++))\ \equiv\ (\mathtt{red}\ (++))\ .\ (\mathtt{map}\ (\mathtt{map}\ f))$$

$$(\mathtt{red}\ \odot)\ .\ (\mathtt{red}\ (++))\ \equiv\ (\mathtt{red}\ \odot)\ .\ (\mathtt{map}\ (\mathtt{red}\ \odot))$$

$(\mathtt{red}\ \odot\ e_\odot)\ .\ (\mathtt{map}\ \mathtt{f})\ \equiv$
$(\mathtt{red}\ \odot\ e_\odot)\ .\ (\mathtt{map}\ ((\mathtt{red}\ \odot\ e_\odot)\ .\ (\mathtt{map}\ \mathtt{f})))\ .\ \mathtt{distr}_p$,
where $\mathtt{distr}_p :: [\alpha] \rightarrow [[\alpha]]_p$,
$(\mathtt{red}\ (\texttt{++})\ [])\ .\ \mathtt{distr}_p = \mathtt{id}$.

Exercises

Exercise 1: Function

h :: Integral a => [[a]] $\rightarrow$ a
h [] = 0
h (x:xs) = (foldr (+) 0 x) + (h xs)

a) Write a list homomorphic implementation of *h*, name it *hh*.

b) Write *hh* in map-reduce style, name it *hMR*

c) Apply the second LH promotion ($\leftarrow$ direction) theorem to optimize it (for example for load-balancing)

d) Use the second LH promotion theorem ($\rightarrow$ direction) to chunk the flattened list into *p* lists, of roughly same number of elements, and to map the computation on each list on one of the *p* processors; and to finally reduce at the end. HINT: use (distr p) :: [a] $\rightarrow$ $[[a]]_p$ to create a list containing *p* lists, and invariant `(red (++)[]).(distr p)=id`.

e) Test all versions in Haskell!

Conclusion

- Imperative language: low-level, “heroic effort”, but effective solutions. Compiler reverse-engineers users sequential optimizations (hard).
- Functional language: parallelism via inherently parallel array combinators, that expose a rich algebra at a higher-level of abstraction.
- Combine the advantages: model the transformations that have proven most useful in the imperative context in a simpler way by using the rich algebra of functional constructs.