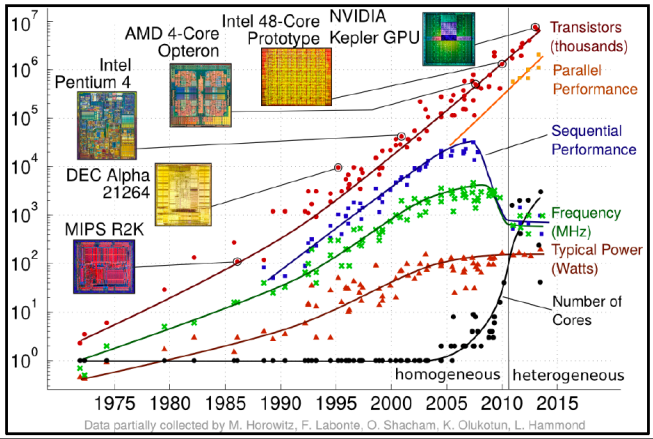
HIGH LEVEL SYNTHESIS FOR FPGAS: EXPLOITING PIPELINE PARALLELISM

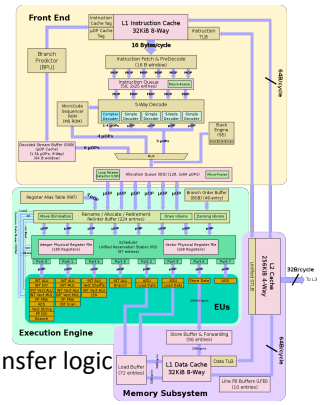
The Architecture Landscape

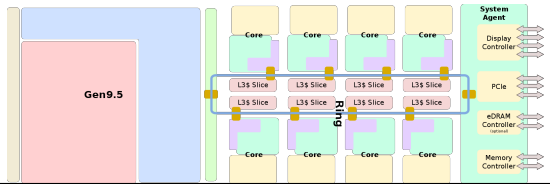
- The world of Transistors has evolved significantly



MultiCores

- Exponential increase in performance
  - Improved organization
  - Increased clock frequency
- Increase in Parallelism
  - Pipelining
  - Superscalar
  - Simultaneous multithreading (SMT)
- Diminishing returns
  - More complexity requires more logic
  - Increasing chip area for coordinating and signal transfer logic
    - Harder to design, make and debug

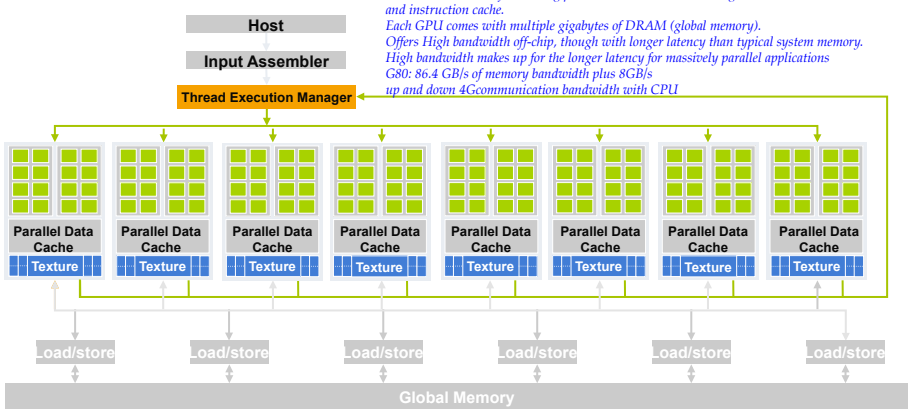




Source: https://en.wikichip.org/wiki/intel/microarchitectures/coffee_lake#Entire_SoC_Overview_.28octa.29

Architecture of a CUDA-capable GPU

Two streaming multiprocessors form a building block
Each has a number of streaming processors that share control logic and instruction cache.
Each GPU comes with multiple gigabytes of DRAM (global memory).
Offers High bandwidth off-chip, though with longer latency than typical system memory.
High bandwidth makes up for the longer latency for massively parallel applications
G80: 86.4 GB/s of memory bandwidth plus 8GB/s
up and down 4Gcommunication bandwidth with CPU



A good application runs 5k to 12k threads. CPU support 2 to 8 threads.

GPUs

- Throughput optimized multicore
  - Performs badly on sequential code
- Sources of parallelism
  - Instruction Level
  - Thread Level
  - Data Level

```
For i = 0 to n-1 do:
   X[i] ← a * X[i]
```

```
Launch n threads:
   X[tid] ← a * X[tid]
```

```
   X ← a * X
```

```
add r3 ← r1, r2
mul r0 ← r0, r1
sub r1 ← r3, r0
```

Parallel

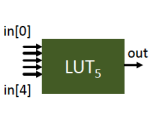
Thread 1 Thread 2

add mul) Parallel

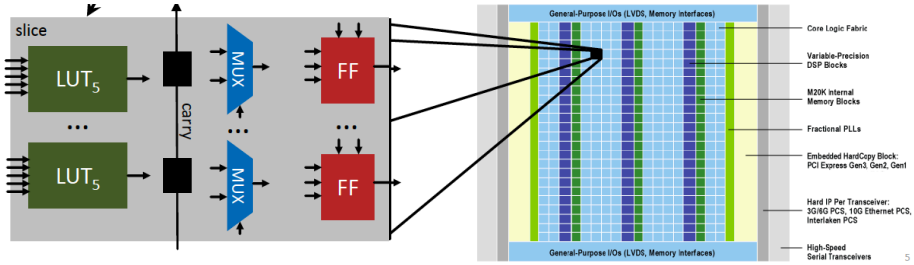
vadd r←a,b

$$\begin{array}{ccc} a_1 & a_2 & a_3 \\ + & + & + \\ b_1 & b_2 & b_3 \\ \hline r_1 & r_2 & r_3 \end{array}$$

Field Programmable Gate Arrays (FPGA)



Address (in[0-4])	Out
00000	0
00001	1
...	...
11111	0

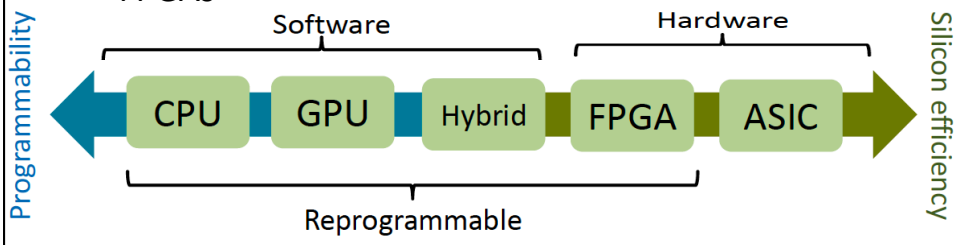


High-Performance FPGA Summaryx

Intel Stratix 10	Virtex Ultrascale	NVidia GPU
14 nm Intel Tri-Gate	16 nm FinFET	12 nm
1 GHz 10 TF single precision	~600 MHz 6,840 DSPs (3.1 TF single prec.)	1455 MHz 5,120 cores (15.7 TF single prec.)
5.5M Logic Elements	2.5M Logic Elements	CUDA programming
4-input LUT, register, carry, etc.	1,182,000 5-input LUTs	On-chip memory:
Block RAM: 28.6 MiB	2,364,000 FFs	Registers: 20.8 MiB
Hardened DRAM controller DDR 4	Block RAM: 9.1 MiB	L1/SM: 7.7 MiB
Various options for memory		L2 Cache: 6.1 MiB
Hyper Flex Interconnect with Regs.		
TDP: 125W (estimated)	TDP: 95 W (Amazon F1 power limit)	TDP: 300W

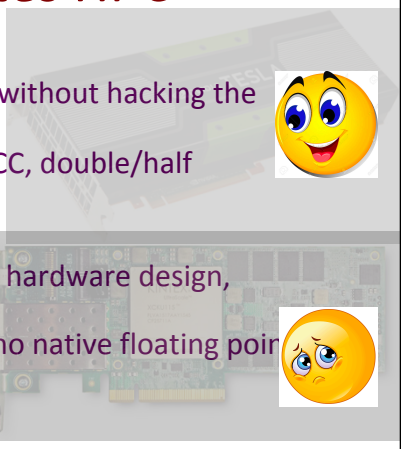
Ease of Programmability vs. Efficiency

- Modern Super computers
  - CPU+GPU
- CGRAs
  - FPGAs



What dominates HPC

- GPUs
  - Productivity: CUDA enabled GPGPU without hacking the graphics pipeline
  - Hardware support: Tesla line with ECC, double/half precision
- FPGAs
  - Productivity: steep learning curve of hardware design, unpolished tools
  - Hardware support: low bandwidth, no native floating poir units
- Recent Developments
  - OpenCL, HLS
  - Intel Startix X and HBM



How do we get Performance

- Massively parallel Computation
  - 250 MHz is often adequate
  - Lower power dissipation
- Parallelism
  - Depth and width of computations performed on the input data
  - Tradeoff between logic, buffering, and time
  - More scope for optimizations
- More optimal data movement
  - Hardwired more often
- Lots of registers
  - Very efficient Pipelines can be easily set up

Programming FPGAs

- Spatial Register Transfer Logic (RTL) Programming
  - Hardware description languages
    - Register Transfer Logic (RTL)
      - VHDL, Verilog, System C, SystemVerilog
    - Very verbose and very low level
      - Every Cycle accounted for
- High-level synthesis
  - Input C/C++/OpenCL is transformed to the spatial paradigm
  - Lift programming from the bit level to the word/datatype level
    - Xilinx and Intel both offer a C/C++ and an OpenCL tool


Bambu

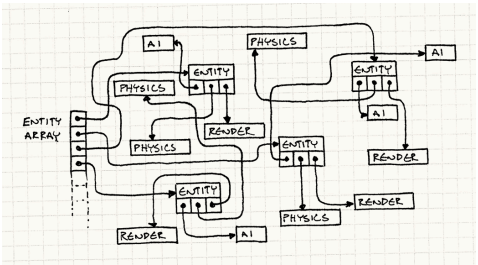
Naneet al.: "A Survey and Evaluation of FPGA High-Level Synthesis Tools", Oct. 2016)

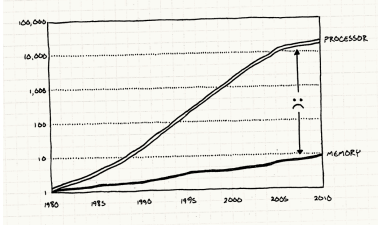
impulse bluespec

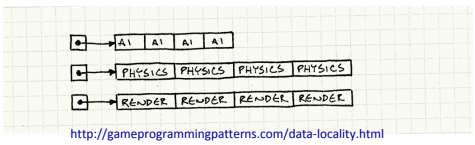

Microsoft Catapult

3 Ls of Modern Computing

- Spatial Locality
- Temporal Locality
- Control Locality





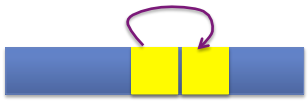


http://gameprogrammingpatterns.com/data-locality.html

Spatial Locality

- *Spatial:* having to do with space -- or in this case, proximity of data
- *Spatial locality:* the principle that data near the data being accessed now will probably be needed soon
- If data item *n* is useful now, then it's likely that data item *n+1* will be useful soon
  - *Data array a accessed with stride 1*
  - *Instructions are accessed in sequence*

```
sum=0
for (i=0;i<=n;i++)
   sum += a[i];
return sum;
```



Temporal Locality

- *Temporal* = having to do with time
- *Temporal locality:* the principle that data being accessed now will probably be accessed again soon
- Useful data tends to continue to be useful
  - *Data variable sum referenced in each iteration*
  - *Same instructions executed in each iteraion*

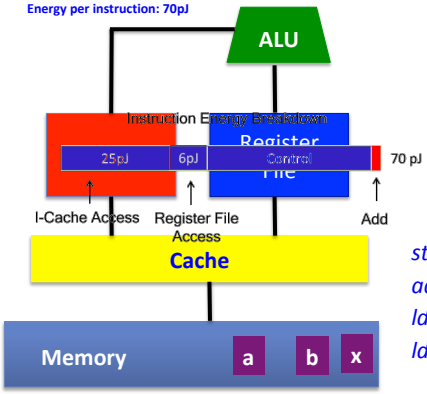
```
sum=0
for (i=0;i<=n;i++)
   sum += a[i];
return sum;
```

Control Locality

Load-store ("von Neumann")

x=a+b

Energy per instruction: 70pJ



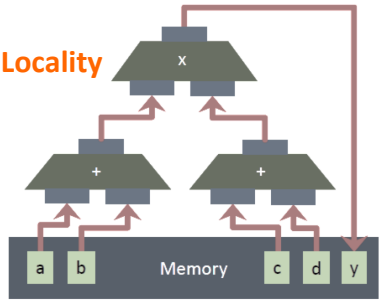
Source: Mark Horowitz, ISSC'14

st x,r3
add r3,r1,r2
ld a,r1
ld b,r2

Static Dataflow ("non von Neumann")

y=(a+b)(c+d)*

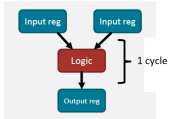
Energy per operation: 1-3pJ



High Level Synthesis

HLS Coding

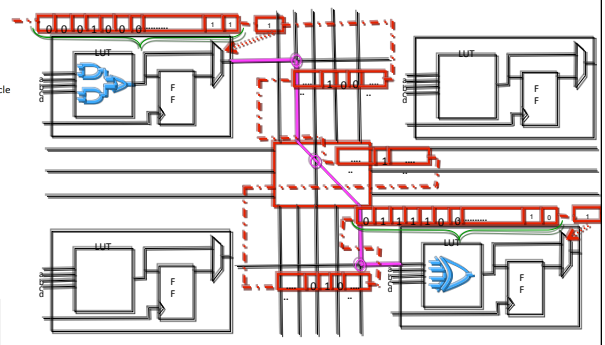
int c = a +b



RTL Coding

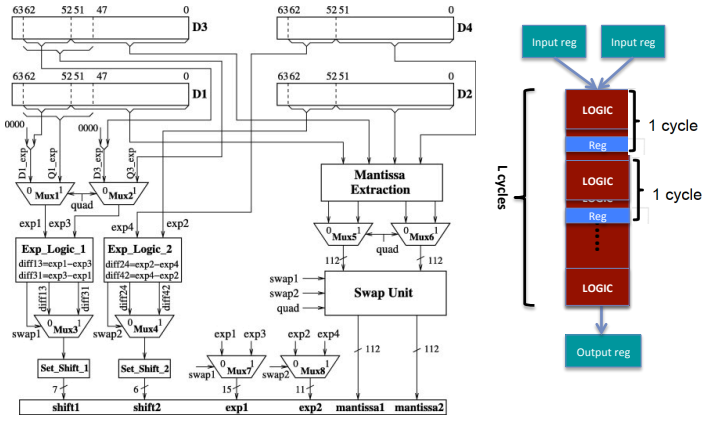
out=in+1

```
always @(posedge clk)
  if (start) begin
      out <= in + 1;
  end
```



HLS

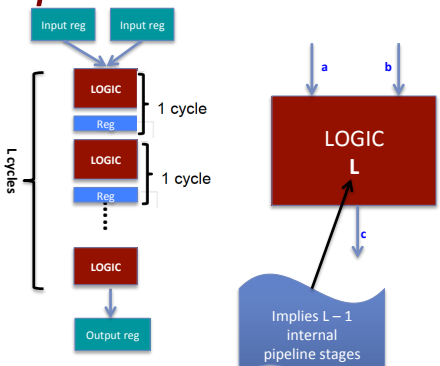
float c = a +b

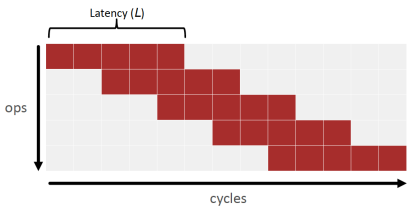


HLS Pipelines

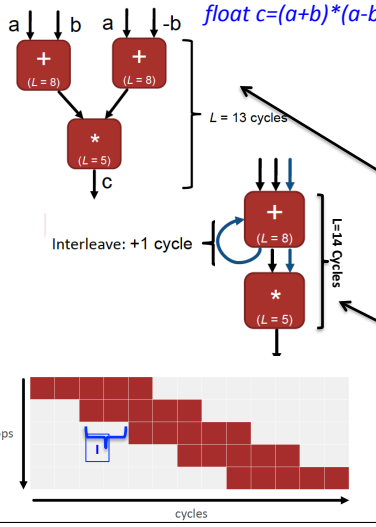
float c = a +b

Pipelines are the Key in HLS





Pipelines Key to Performance



- Initiation interval
  - In addition to latency (L), we introduce the property initiation interval
    - "II", here I
  - No. of cycles before we can accept new inputs
- Implementation 1 can accept all 4 inputs
  - L = 13 cycles
  - I = 1 cycle
  - 2 adds, 1 mult

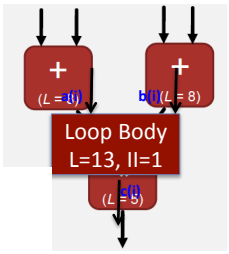
  3 op/1 cycle
- Implementation 2
  - L = 14 cycles
  - I = 2 cycles
  - 1 add, 1 mult

  3 op/2 cycle

Throughput is halved!

Pipelines --- with Loops

```
for (int i = 0; i < N; ++i) {
#pragma HLS PIPELINE II=1
c[i] = (a[i] + b[i]) * (a[i] - b[i]);
}
```



1 iteration	13 + 1 = 14 cycles
10 iterations	13 + 10 = 23 cycles
N iterations	13 + N cycles

Loop iterations affect the runtime additively, regardless of body content

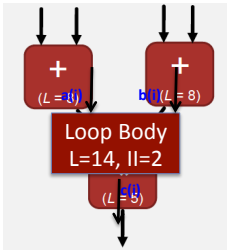
Pipelines --- with Loops

```
for (int i = 0; i < N; ++i) {
#pragma HLS PIPELINE II=2
c[i] = (a[i] + b[i]) * (a[i] - b[i]);
}
```

1 iteration 14 + 2 = 16 cycles
10 iterations 14 + 20 = 34 cycles
N iterations 14 + 2N cycles

$L_{tot} = L + II*N$

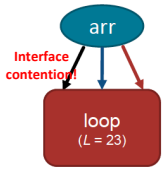
*Initiation interval paid at **every** iteration*



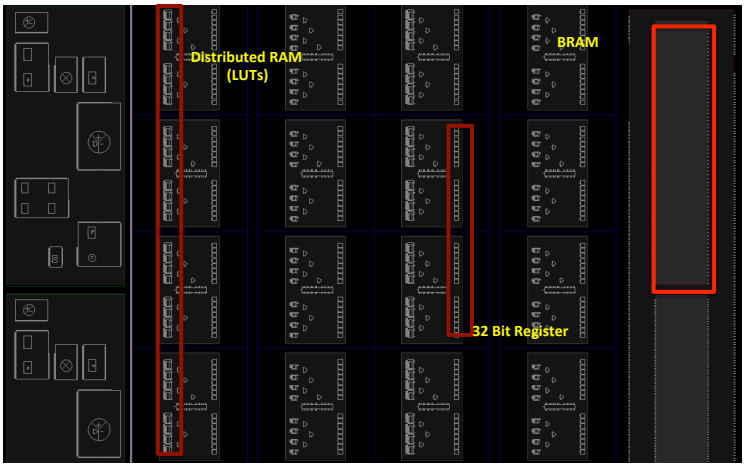
Lets Look at How FPGAs help/Suffer

- Initiation Interval essentially results in a Pipeline Stall
  - So why not always have II=1
- Intra-iteration:
  - Multiple accesses to the same interface
- Inter-iteration
  - Data dependencies
- Low throughput requirements
  - input only received every 16 cycles

```
for (int i = 1; i < N - 1; ++i) {
#pragma HLS PIPELINE II=1 II=3
res[i] = 0.3333 * (arr[i-1] + arr[i] + arr[i+1]);
}
```



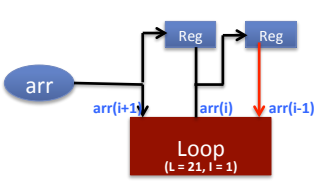
FPGA Memory



Insert Registers

```
for (int i = 1; i < N - 1; ++i) {
res[i] = 0.3333 * (arr[i-1] + arr[i] + arr[i+1]);
}
```

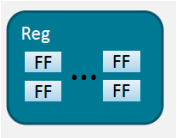
So How is relevant to HLS and FPGAs



Stage 1

Stage 2

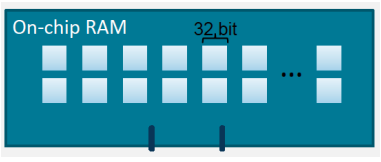
Stage 3



4 Byte = 32 bit · 1

D_{reg}=1

W_{reg}=32



D_{RAM}=1024

W_{RAM}=32

4096 Byte = 32 bit · 1024

Memory: depth (D) and width (W)

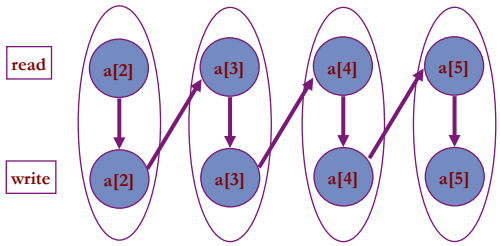
Transformations

- Pipeline-enabling transformations
  - Transposition
  - Interleaving
  - Cross-input pipelining
  - Inlining
  - Cyclic buffering
  - Pipelined loop flattening/coalescing
  - Pipelined loop fusion
- Scalability transformations
  - Vectorization
  - Replication
  - Streaming dataflow
  - Tiling
- Secondary transformations
  - Memory access extraction
  - Memory oversubscription
  - Memory striping
  - Type demotion
- Optimization Goals
  - Perfect Pipelining
    - I=1
    - Maximum Throughput
  - Scaling/Folding
    - Fold N by scaling up the parallelism of the design
  - Saturation
    - Saturate pipelines for the majority of the runtime
      - No Stalls

Dependences

- Scalar Variables
  - True Dependence
    - A =
    - = A
  - Anti Dependence
    - = A
    - A =
  - Output Dependence
    - A =
    - A =
  - Input Dependence
    - = A
    - = A
- Loop Variables

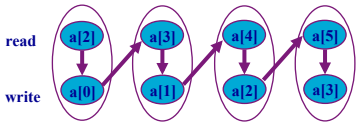
for i= 2, 5
a[i] = a[i] + 3



Dependence in Loops

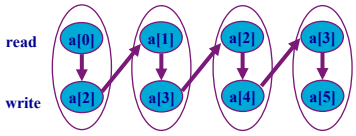
Array Anti-dependence

for i= 2, 5
a[i-2] = a[i] + 3



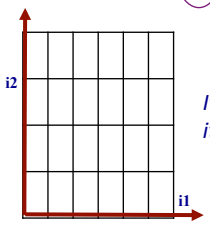
Array True-dependence

for i= 2, 5
a[i] = a[i-2] + 3



Iteration Space

```
for i1 = 0, 5
  for i2 = 0, 3
    a[i1,i2] =  3
```



Iteration is represented as coordinates in iteration space

Loop Carried Dependence

- There exists a dependence from statement S1 to S2 in a common nest of loops iff there exist two iteration vectors i and j such that
  - i < j or i = j and there is a path from S1 to S2 in the body of the loop
  - S1 accesses memory location M on iteration i and S2 accesses M on iteration j
  - one of these accesses is a write
- Loop Carried Dependence
  - Statement S_2 has a loop-carried dependence on statement S_1 if and only if S_1 references location M on iteration **i**, S_2 references M on iteration **j**

for i= 2, 5
a[i+1] = f[i] + 3
f[i+1]=a[i]

Iteration Space Transposition

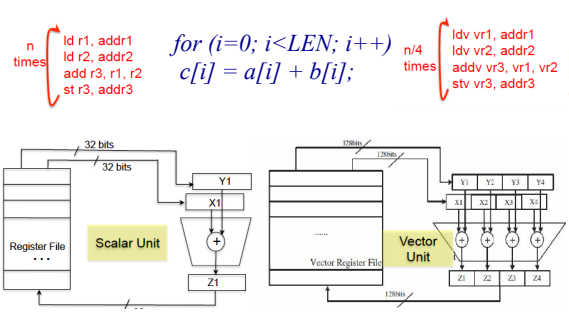
- Modified Matrix Multiplication
  - C=A*B+C
    - Multiplication of elements of A and B can be pipelined
    - Addition on Line 8 requires the result of the addition in the previous iteration of the loop

```
1 for (int n = 0; n < N; ++n)
2
3   for (int p = 0; p < P; ++p) {
4     auto acc = C[n][p];
5     #pragma PIPELINE
6     for (int m = 0; m < M; ++m)
7       // Loop-carried dependency
8       acc += A[n][m] * B[m][p];
9     C[n][p] = acc;
10  }
```

```
1 for (int n = 0; n < N; ++n) {
2   float acc[P]; // Uninitialized
3   for (int m = 0; m < M; ++m)
4     auto a = A[n][m];
5     #pragma PIPELINE
6     for (int p = 0; p < P; ++p) {
7       auto prev = (m == 0) ? C[n][p] : acc[p];
8       acc[p] = prev + a * B[m][p]; }
9   for (int p = 0; p < P; ++p)
10    C[n][p] = acc[p]; }
```

Vectorization

- Exploit SIMD parallelism with HLS
  - Partially unrolling loop nests in pipelined sections
  - Can be directly applied to the inner loop



Vectorization by strip-mining

```
1 for (int i = 0; i < N / W; ++i)
2   #pragma UNROLL // Fully unroll inner loop
3   for (int w = 0; w < W; ++w)
4     C[i*W + w] = A[i*W + w] * B[i*W + w];
```


Vectorization by partial unrolling

```
1 #pragma UNROLL W // By factor W
2 for (int i = 0; i < N; ++i)
3   C[i] = A[i] * B[i];
```

Type Demotion

- Demote Data Types
  - Less expensive alternatives
  - Must meet precision Requirements
    - Reduce resource and energy consumption
    - Bandwidth requirements
    - Operation latency
  - Use less Resources
    - Compute Bound
      - » Floating point to fixed point
      - » Use Native Data types (16 bit for Xilinx)
    - Bandwidth Bound
      - » Performance improves by the the same factor that the size of the data type can be reduced
    - Latency Bound
      - » Floating point ops → multiple cycles: Integer ops→ 1 cycle

Software Transformations In HLS

CPU transformation	In HLS
Loop interchange [2, 36]	Used to resolve loop carried dependencies throughout Section 2.
Strip-mining [77]	Central component of many HLS transformations, including accumulation interleaving (Section 2.2), vectorization (Section 3.1), replication (Section 3.2), and tiling (Section 3.4).
Loop tiling [36, 40]	
Cycle shrinking [56]	
Loop distribution/fission [35, 36]	Useful for separating differently scheduled computations to allow pipelining (see Section 3.3).
Loop fusion [36, 79, 83]	Used for merging pipelines (see Section 2.7).
Loop unrolling [18]	Essential tool for scaling up performance by generating more computational hardware (Section 3.1 and 3.2).
Software pipelining [39]	Used by the HLS tool to schedule loop bodies according to the interdependencies of operations.
Loop coalescing/flattening [55]	Used to save pipeline drains in nested loops (Section 2.6).
Loop collapsing	
Reduction recognition	Prevent loop-carried dependencies in accumulation codes (Section 2.1 and 2.3).
Loop idiom recognition	Relevant for HLS backends, for example to recognize sliding-window buffers (Section 2.5) in Intel OpenCL [72].
Procedure inlining	Required to pipeline code sections with function calls (Section 2.4).
Procedure cloning	Every occurrence of a function is always specialized to all variables that can be statically inferred.
Loop unswitching [17]	Often the **opposite** is beneficial (see Section 2.6 and 2.7).
Loop peeling	Often the **opposite** is beneficial to allow coalescing (Section 2.6).
Graph partitioning	Streaming is central to hardware algorithms (Section 3.3).
SIMD transformations	Covered in Section 3.1.

Reference

- Paper Reference
  - Transformations of High-Level Synthesis Codes for High-Performance Computing
    - https://arxiv.org/abs/1805.08288
- Slides are mostly derived from
  - https://spcl.inf.ethz.ch/Teaching/2018-sc/

Interleaving accumulations to eliminate the loop-carried dependency

```
1 for (int i = 0; i < N; ++i) {
2   Vec<double, 3> acc;
3   Vec<double, 3> s0 = s[i];
4
5
6
7   #pragma PIPELINE
8   for (int j = 0; j < N; ++j)
9     acc += Force(s0, s[j], m[j]);
10
11  v[i] = v[i] + dt * acc;
12  s[i] = s0   + dt * v[i]; }
```

(a) N-body code with loop-carried dependency.

```
1 for (int i = 0; i < N / K; ++i) {
2   Vec<double, 3> acc[K];
3   Vec<double, 3> s0[K];
4   for (int k = 0; k < K; ++K)
5     s0[k] = s[i*K + k];
6   for (int j = 0; j < N; ++j)
7     #pragma PIPELINE
8     for (int k = 0; k < K; ++j)
9       acc[k] += Force(s0[k], s[j], m[j]);
10  for (int k = 0; k < K; ++K) {
11    v[i*K + k] += dt * acc;
12    s[i*K + k] += dt * v[i*K + k]; }}
```

(b) Strip-mine outer loop to interleave K accumulations.