US 20200257510A1

(19) United States

(12) Patent Application Publication

Deodhar et al.

(10) Pub. No.: US 2020/0257510 A1

(43) Pub. Date: Aug. 13, 2020

# (54) AUTOMATIC COMPILER DATAFLOW OPTIMIZATION TO ENABLE PIPELINING OF LOOPS WITH LOCAL STORAGE REQUIREMENTS

(71) Applicant: **Intel Corporation**, Santa Clara, CA (US)

(72) Inventors: **Rajiv Deodhar**, Phoenix, AZ (US); **Sergey Dmitriev**, Novosibirsk (RU); **Daniel Woodworth**, Hudson, MA (US); **Rakesh Krishnaiyer**, Milpitas, CA (US); **Kent Glossop**, Nashua, NH (US); **Arvind Sudarsanam**, Lexington, MA (US)

(21) Appl. No.: **16/863,315**

(22) Filed: **Apr. 30, 2020**

**Publication Classification**

(51) **Int. Cl.**

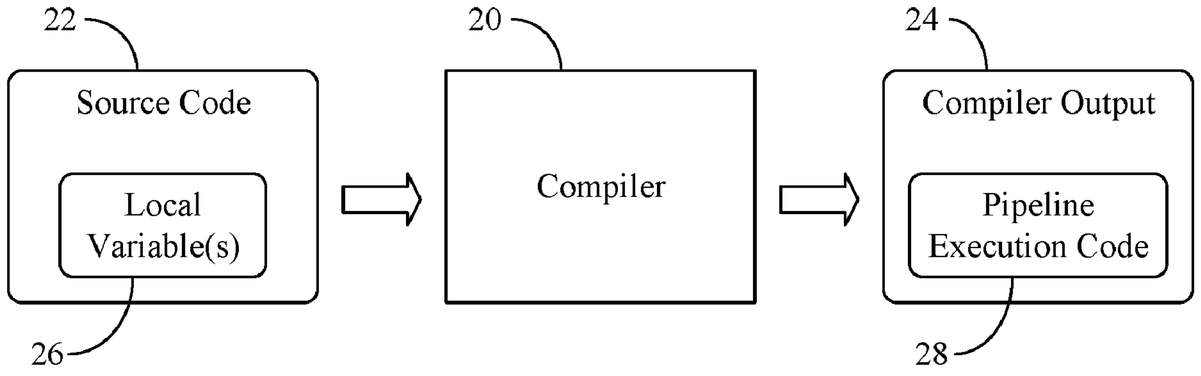
| | |
|---|---|
| ***G06F 8/41*** | (2006.01) |
| ***G06F 9/38*** | (2006.01) |

(52) **U.S. Cl.**

CPC ............ ***G06F 8/433*** (2013.01); ***G06F 9/3838*** (2013.01); ***G06F 9/381*** (2013.01); ***G06F 9/3867*** (2013.01)

(57) **ABSTRACT**

Systems, apparatuses and methods may provide for technology that detects one or more local variables in source code, wherein the local variable(s) lack dependencies across iterations of a loop in the source code, automatically generate pipeline execution code for the local variable(s), and incorporate the pipeline execution code into an output of a compiler. In one example, the pipeline execution code includes an initialization of a pool of buffer storage for the local variable(s).



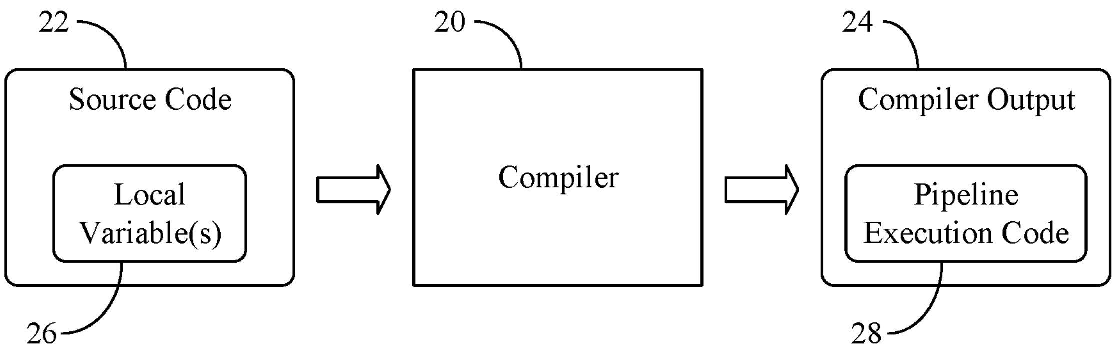


FIG. 1

```
int f(int n, int x, int y, int* a)
{
  for (int i=0; i<n; i++) {
    int b[100];
    for (int j=x; j<y; j++) {
      b[j] = i;
    }
    a[i] = a[i] + b[6];
  }
  return a[5];
}
```

FIG. 2A

32

```
static void SpGEMMKernel(int ibegin, int iend, int n, int k,
  const int *rowptrA, const int *colidxA, const FLOAT *valA,
  const int *rowptrB, const int *colidxB, const FLOAT *valB,
  const int *rowptrC, int *colidxC, FLOAT *valC)
{
    #pragma omp parallel for dataflow(pipeline(8))
      for (int i = ibegin; i < iend; i++) {
          // Private variable
          FLOAT spa[n];
          memset(spa, 0, sizeof(FLOAT) * n);
          int startA = rowptrA[i];
          int endA = rowptrA[i + 1];
          for (int j = startA; j < endA; j++) {
              int idxA= colidxA[j];
              double vA = valA[j];
              int startB= rowptrB[idxA];
              int endB= rowptrB[idxA + 1];
              for (int k = startB; k < endB; k++) {
                  int idxB = colidxB[k];
                  double vB = valB[k];
                  spa[idxB] += vA * vB;
              } // for (int k = startB; k < endB; k++)
          } // for (int j = startA; j < endA; j++)
          int idxC = rowptrC[i];
          for (int j = 0; j < n; j++) {
              if (spa[j] != 0.0) {
                  colidxC[idxC] = j;
                  valC[idxC] = spa[j];
                  idxC++;
                  spa[j] = 0.0;
              }
          } // for (int j = 0; j < n; j++)
      } // for (int i = 0; i < m; i++)
}
```

FIG. 2B

34

```
int f(int n, int x, int y, int* a)
{
  int b[100];
  #pragma omp parallel for private(b)
  for (int i=0; i<n; i++) {
    for (int j=x; j<y; j++) {
      b[j] = i;
    }
    a[i] = a[i] + b[6];
  }
  return a[5];
}
```

**FIG. 2C**

36

```
int f(int n, int x, int y, int* a)
{
  #pragma omp parallel for
  for (int i=0; i<n; i++) {
#if CONSTANT_SIZE
    // Constant-sized array, dynamically allocated
    int* b = (int *)malloc(100 * sizeof(int));
#else
    // Variable-sized array, dynamically allocated
    int* b = (int *)malloc(n * sizeof(int));
#endif
    for (int j=x; j<y; j++) {
      b[j] = i;
    }
    a[i] = a[i] + b[6];
    …
  }
  return a[5];
}
```

**FIG. 2D**

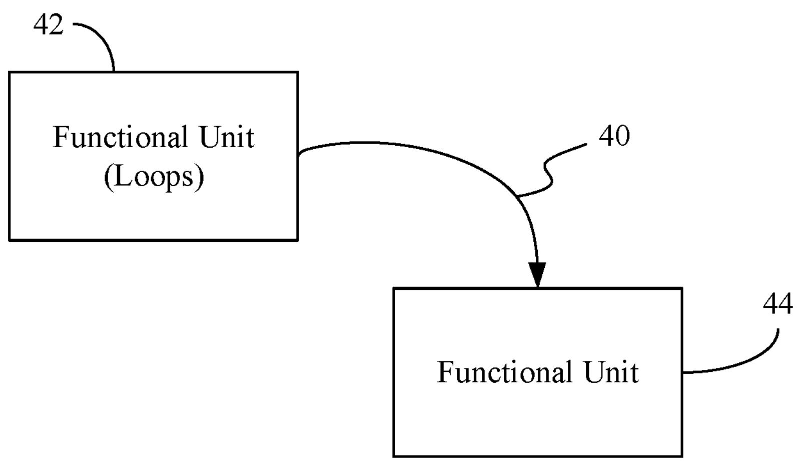

42
Functional Unit (Loops)
40
Functional Unit
44

FIG. 3

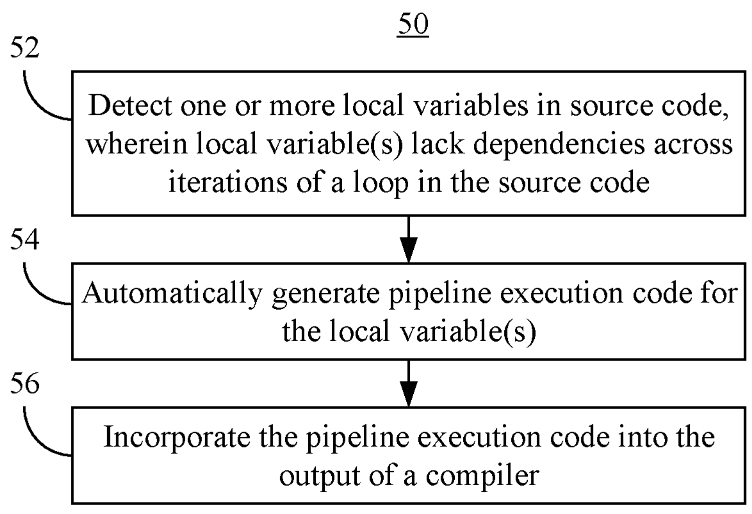

50
52
Detect one or more local variables in source code, wherein local variable(s) lack dependencies across iterations of a loop in the source code
54
Automatically generate pipeline execution code for the local variable(s)
56
Incorporate the pipeline execution code into the output of a compiler

FIG. 4

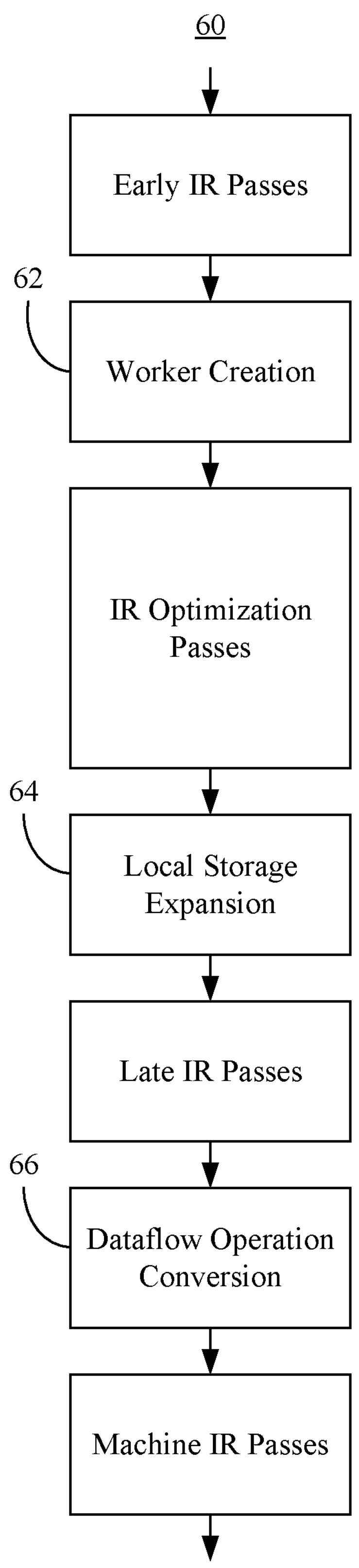

60
Early IR Passes
62
Worker Creation
IR Optimization Passes
64
Local Storage Expansion
Late IR Passes
66
Dataflow Operation Conversion
Machine IR Passes

FIG. 5

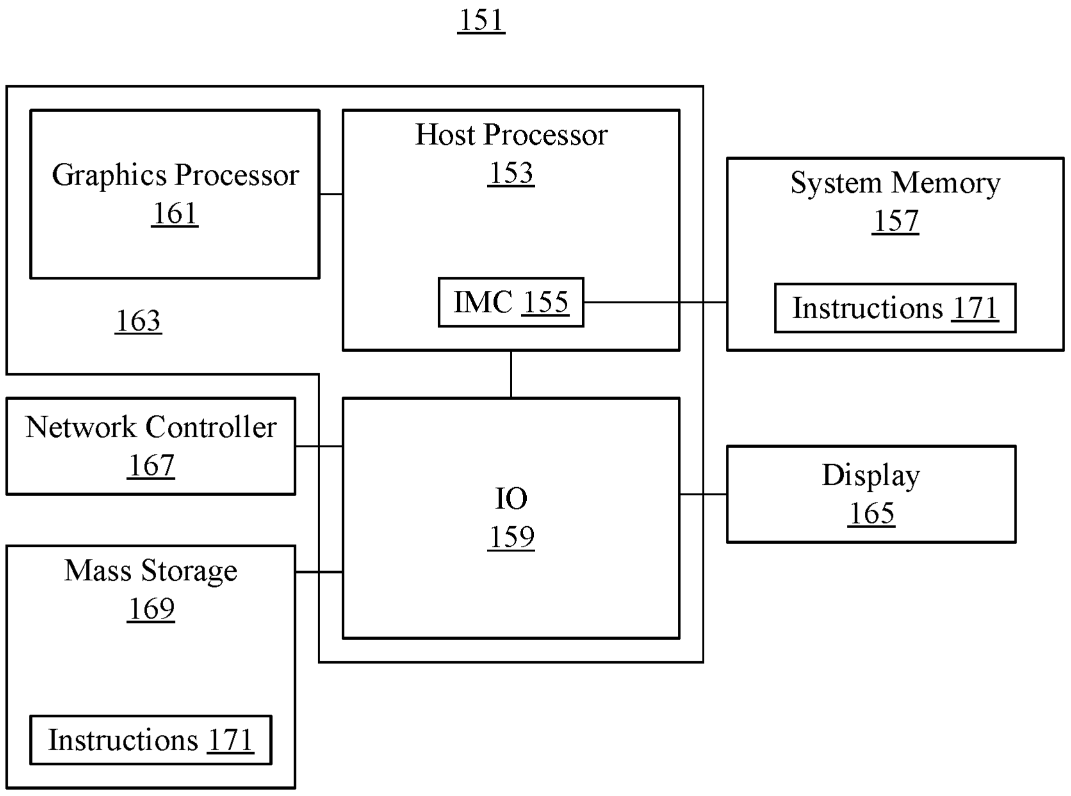

151
Graphics Processor
161
Host Processor
153
IMC 155
163
System Memory
157
Instructions 171
Network Controller
167
IO
159
Display
165
Mass Storage
169
Instructions 171

**FIG. 6**

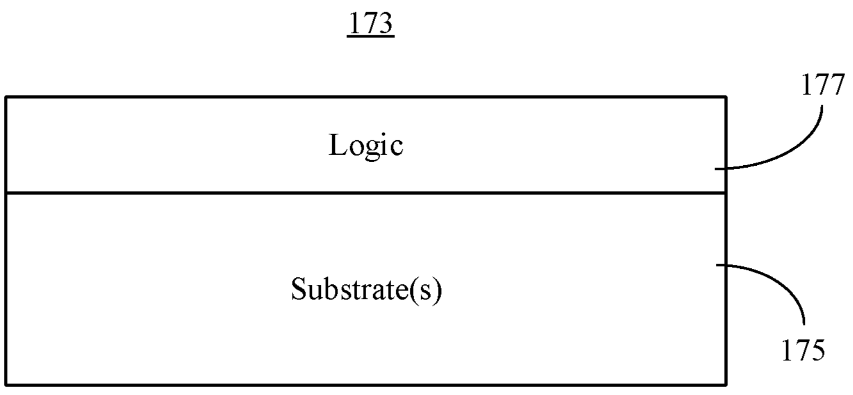

173
Logic
177
Substrate(s)
175

**FIG. 7**

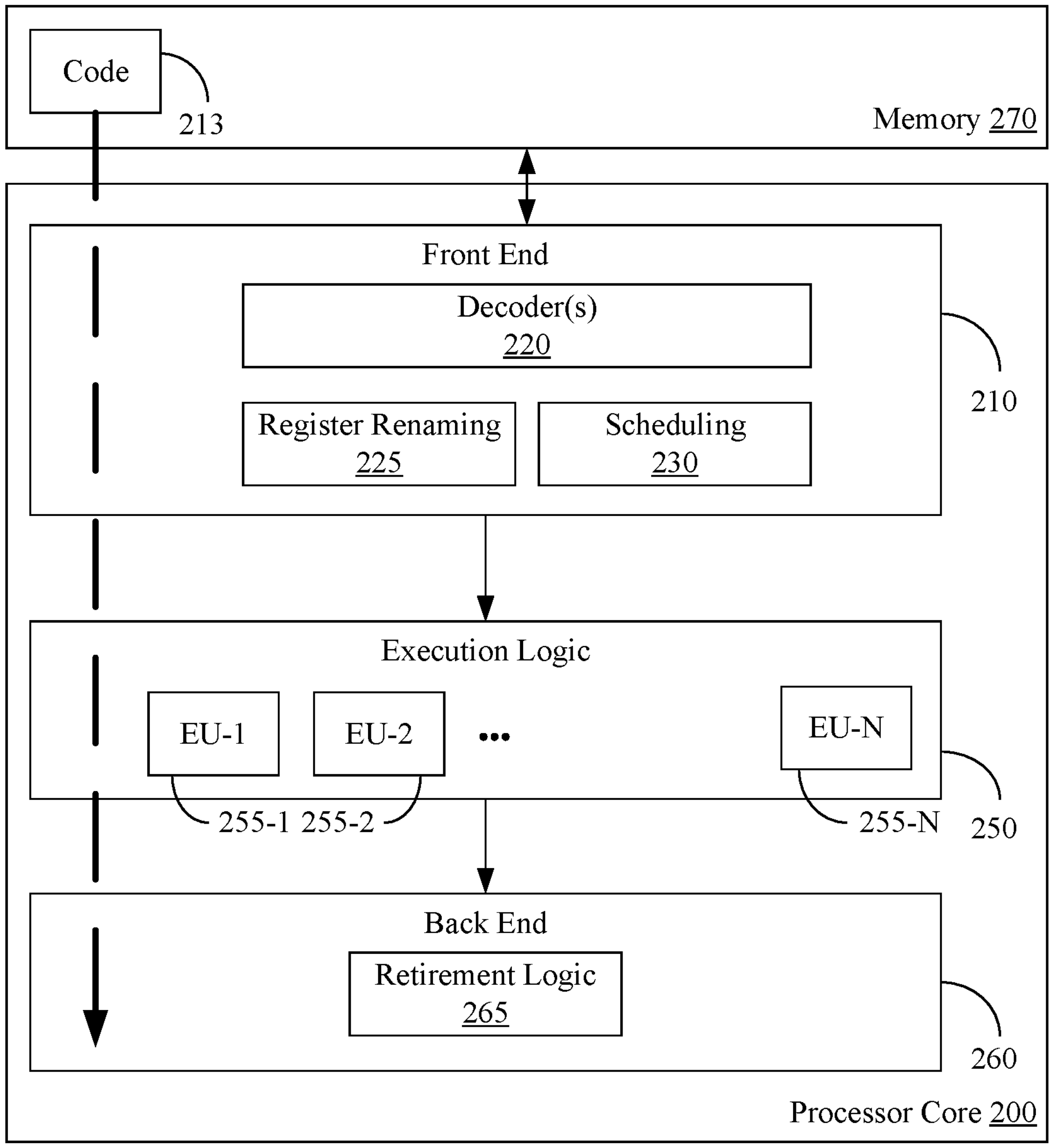
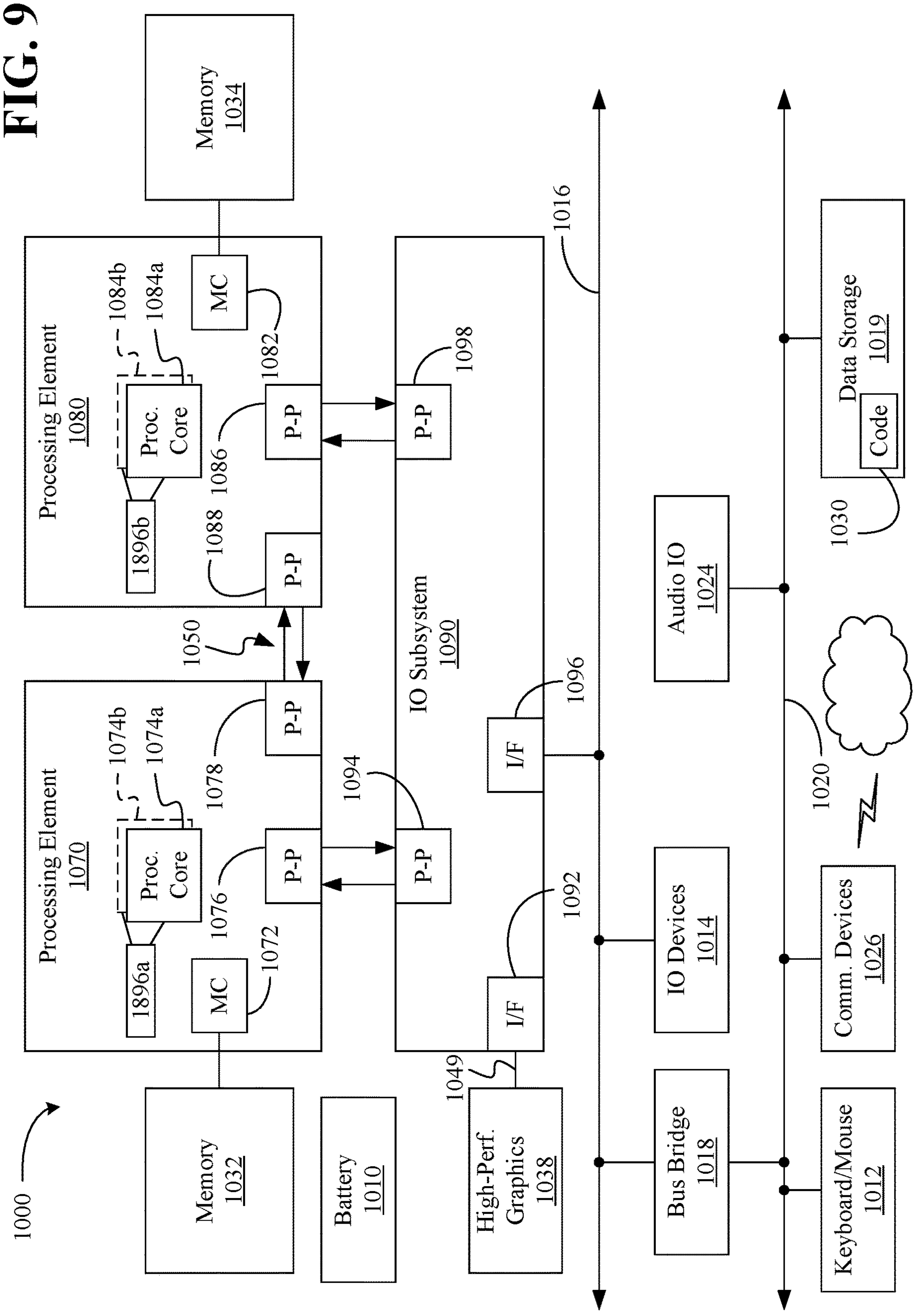

Code
213
Memory 270
Front End
Decoder(s) 220
Register Renaming 225
Scheduling 230
210
Execution Logic
EU-1
EU-2
•••
EU-N
255-1
255-2
255-N
250
Back End
Retirement Logic 265
260
Processor Core 200

FIG. 8

FIG. 9


1000
Memory 1032
Memory 1034
Battery 1010
Processing Element 1070
Processing Element 1080
Proc. Core
1074a
1074b
1084a
1084b
1896a
1896b
MC
1072
MC
1082
P-P
1076
1078
1086
1088
1050
1094
1098
IO Subsystem 1090
I/F
1092
1096
1049
High-Perf. Graphics 1038
1016
Bus Bridge 1018
IO Devices 1014
Audio IO 1024
1020
Keyboard/Mouse 1012
Comm. Devices 1026
Data Storage 1019
Code
1030

# AUTOMATIC COMPILER DATAFLOW OPTIMIZATION TO ENABLE PIPELINING OF LOOPS WITH LOCAL STORAGE REQUIREMENTS

## COPYRIGHT NOTICE

## TECHNICAL FIELD

**[0002]** Embodiments generally relate to compilers. More particularly, embodiments relate to automatic compiler dataflow optimizations to enable pipelining of loops with local storage requirements.

## BACKGROUND

**[0003]** Dataflow graphs may be used to model computer source code in terms of the dependencies between individual operations performed by the code. A compiler may transform the source code into the dataflow graph, which is typically executed by accelerator hardware such as a field programmable gate array (FPGA), configurable spatial accelerator (CSA), or other dataflow architecture. While the accelerator hardware may be useful when dealing with high performance computing (HPC) and/or data center applications that operate on relatively large data arrays and structures, there remains considerable room for improvement. For example, if the operations of the source code involve the execution of loops that internally declare "private" variables for large data arrays, the ability to hold (e.g., "registerize") the underlying data in the internal channels (e.g., communication arcs, buffers, latency insensitive channels/LICs, etc.) of the accelerator may be limited. As a result, the private variables may be treated as purely memory-based variables, which may cause performance losses.

## BRIEF DESCRIPTION OF THE DRAWINGS

**[0004]** The various advantages of the embodiments will become apparent to one skilled in the art by reading the following specification and appended claims, and by referencing the following drawings, in which:

**[0005]** FIG. **1** is a block diagram of an example of a compiler output according to an embodiment;

**[0006]** FIG. **2**A is a source code listing of an example of a loop with fixed-size local storage according to an embodiment;

**[0007]** FIG. **2**B is a source code listing of an example of a loop with runtime-varying local storage according to an embodiment;

**[0008]** FIG. **2**C is a source code listing of an example of a loop with an explicitly designated private variable according to an embodiment;

**[0009]** FIG. **2**D is a source code listing of an example of a loop with a dynamically allocated local variable according to an embodiment;

**[0010]** FIG. **3** is a block diagram of an example of a communication arc in a dataflow graph according to an embodiment;

**[0011]** FIG. **4** is a flowchart of an example of a method of operating a compiler according to an embodiment;

**[0012]** FIG. **5** is a block diagram of an example of a compiler according to an embodiment;

**[0013]** FIG. **6** is a block diagram of an example of a performance-enhanced computing system according to an embodiment;

**[0014]** FIG. **7** is an illustration of an example of a semiconductor apparatus according to an embodiment;

**[0015]** FIG. **8** is a block diagram of an example of a processor according to an embodiment; and

**[0016]** FIG. **9** is a block diagram of an example of a multi-processor based computing system according to an embodiment.

## DESCRIPTION OF EMBODIMENTS

**[0017]** Turning now to FIG. **1**, a compiler **20** is shown, where the compiler **20** automatically transforms source code **22** into an output **24** that is executable by a dataflow architecture such as, for example, an FPGA, CSA, and so forth. In an embodiment, the source code **22** is written in a high-level language such as, for example, C, C++, or Fortran augmented by parallel annotations (e.g., OpenMP parallel pragmas) to achieve runtime parallelism in the dataflow architecture. The source code **22** may generally use loops to perform various operations. Indeed, the runtime performance of applications may be dominated by the time spent in executing loops to perform tasks. On a dataflow architecture such as CSA, the performance of parallel loops may be accelerated by a) creating multiple copies of the loop bodies (e.g., "workers"), b) executing the workers in parallel, and c) pipelining execution of the workers.

**[0018]** In the illustrated example, the source code **22** contains one or more local variables **26** (e.g., private variables), which lack dependencies across iterations of the loops in the source code **22**. As will be discussed in greater detail, such a variable might occur naturally when declared inside a loop. In an embodiment, the local variable(s) **26** are occasionally used for relatively large data arrays. To improve the throughput of the loops containing the local variable(s) **26** in such a case, the illustrated compiler **20** generates pipeline execution code **28** for the local variable(s) **26** and incorporates the pipeline execution code **28** into the output **24** of the compiler **20**. Thus, the illustrated local variables are allocated in a way that each loop iteration gets its own copy, thereby permitting pipelined execution. As already noted, pipelining execution of the workers may significantly enhance performance.

**[0019]** FIG. **2**A shows source code **30** containing a loop (e.g., "for (int i=0; i<n; i++") that declares a variable "b", which may be considered a local variable because it lacks dependencies across iterations of the loop. In the illustrated example, the variable has a fixed size (e.g., an array of 100 integers). Thus, the local storage requirements of the variable b are fixed and statically known to the compiler. The illustrated source code **30** may be readily substituted for the source code **22** (FIG. **1**), already discussed. Accordingly, pipeline execution code may be automatically generated for the illustrated local variable.

**[0020]** FIG. **2**B shows source code **32** containing a loop (e.g., "for (int i=ibegin; i<iend; i++)") that declares a vari-

able "spa", which also lacks dependencies across iterations of the loop and is considered a local variable. In the illustrated example, the size of the variable varies and is only known at runtime. The illustrated source code **32** may be readily substituted for the source code **22** (FIG. **1**), already discussed. Accordingly, pipeline execution code may be automatically generated for the illustrated local variable.

**[0021]** FIG. **2**C shows source code **34** containing a loop (e.g., "for (int j=x; j<y; j++)") that uses a variable "b", where the variable b is explicitly designated as a private variable (e.g., using the "private" clause). Other explicit clauses such as "firstprivate", "lastprivate", "reduction", etc., may also be used. In the illustrated example, the variable has a fixed size (e.g., an array of 100 integers). Thus, the local storage requirements of the variable b are fixed and statically known to the compiler. The illustrated source code **34** may be readily substituted for the source code **22** (FIG. **1**), already discussed. Accordingly, pipeline execution code may be automatically generated for the illustrated local variable.

**[0022]** FIG. **2**D shows source code **36** containing a loop (e.g., "for (int i=0; i<n; i++)") that dynamically allocates memory for a variable "b" from within the loop. In the illustrated example, the variable is a local variable that lacks dependencies across iterations of the loop and the size of the variable may either remain constant or vary. The illustrated source code **36** may be readily substituted for the source code **22** (FIG. **1**), already discussed. Accordingly, pipeline execution code may be automatically generated for the illustrated local variable.

**[0023]** Turning now to FIG. **3**, a communication arc **40** (e.g., LIC) between a first functional unit **42** (e.g., node) in a dataflow graph and a second functional unit **44** in the dataflow graph is shown. In the illustrated example, the functional units **42** and **44** are used to perform operations in a loop on data associated with local variables. In an embodiment, the communication arc **40** includes buffer storage (not shown) such as, for example, one or more line buffers, FIFO (first in first out) buffers, etc., which may be used to hold values that enable apportioning data associated with local variables in the loop to different loop iterations.

**[0024]** FIG. **4** shows a method **50** of operating a compiler. The method **50** may generally be implemented in a compiler such as, for example, the compiler **20** (FIG. **1**), already discussed. More particularly, the method **50** may be implemented in one or more modules as a set of logic instructions stored in a machine- or computer-readable storage medium such as random access memory (RAM), read only memory (ROM), programmable ROM (PROM), firmware, flash memory, etc., in configurable logic such as, for example, programmable logic arrays (PLAs), FPGAs, complex programmable logic devices (CPLDs), in fixed-functionality logic hardware using circuit technology such as, for example, application specific integrated circuit (ASIC), complementary metal oxide semiconductor (CMOS) or transistor-transistor logic (TTL) technology, or any combination thereof.

**[0025]** For example, computer program code to carry out operations shown in the method **50** may be written in any combination of one or more programming languages, including an object oriented programming language such as JAVA, SMALLTALK, C++ or the like and conventional procedural programming languages, such as the "C" programming language or similar programming languages. Additionally, logic instructions might include assembler instructions, instruction set architecture (ISA) instructions, machine instructions, machine dependent instructions, microcode, state-setting data, configuration data for integrated circuitry, state information that personalizes electronic circuitry and/or other structural components that are native to hardware (e.g., host processor, central processing unit/CPU, microcontroller, etc.).

**[0026]** Illustrated processing block **52** provides for detecting one or more local variables in source code, wherein the local variable(s) lack dependencies across iterations of a loop in the source code. The source code may be associated with a communication channel as, for example, the communication arc **40** (FIG. **3**) in a dataflow graph. In an embodiment, block **52** includes automatically parsing and/or searching the source code for loops with fixed-size local storage (e.g., as in FIG. **2**A), runtime-varying local storage (e.g., as in FIG. **2**B), explicitly designated private variables (e.g., as in FIG. **2**C), dynamically allocated local variables (e.g., as in FIG. **2**D), and so forth. Moreover, block **52** may be conducted after a registerization of the source code.

**[0027]** Block **54** automatically generates (e.g., in response to the detection of the one or more local variables) pipeline execution code for the local variable(s). As will be discussed in greater detail, block **54** may include generating executable instructions to initialize a pool of buffer storage for the local variable(s), define a pipeline depth, and define a plurality of tokenized slots in the initialized pool of buffer storage. In one example, the initialized pool of buffer storage is greater than the local storage amount corresponding to a single iteration of the loop. Moreover, each tokenized slot may correspond to a pipelined iteration of the loop. Illustrated block **56** incorporates the pipeline execution code into the output of the compiler. The method **50** therefore enhances performance by enabling the pipelining of loops containing private data, which improves throughput. Indeed, the overall cycles needed to execute a loop may be significantly less than the product of static loop cycles (e.g., the number of cycles needed to execute one iteration of the loop) and the loop iteration count.

**[0028]** FIG. **5** shows a compiler **60** that may implement one or more aspects of the method **50** (FIG. **4**), already discussed. Additionally, the compiler **60** may be readily substituted for the compiler **20** (FIG. **1**), already discussed. In general, the compiler **60** enables pipelined execution of loops containing local variables and may be explained with reference to a piece of sample source code and compiler-generated pseudo-code. For further reference, the end of this disclosure includes actual intermediate representation (IR) results using an LLVM compiler for a similar sample before and after the principal compiler transformations described herein.

**[0029]** Using dynamically allocated local storage in a loop as an example, with a constant array size of 100 chosen for simplicity, it may be assumed that the compiler **60** selects two workers for the loop and chooses a pipeline depth of three for each worker loop.

**[0030]** An OpenMP language extension may also be implemented to allow explicit control over worker creation and pipeline depth. Such an extension may be considered optional.

**[0031]** The OpenMP language extension is:

```
#pragma omp ...
dataflow([num_workers[(<n>)]],
[static](<chunksize>)]],
[pipeline](<depth>)]])
   for-loops
```

**[0032]** The pipeline(depth) sub-clause specifies how many loop iterations are to be allowed to execute concurrently. The num workers and static clauses specify how many workers to create and the way to distribute the loop iterations across the workers. Other parallel annotation languages and/or APIs (application programming interfaces) such as OpenACC, OpenCL, SYCL, etc., may also be used.

**[0033]** The solution for correctly handling private variables in pipelined loops may span many passes in the compiler **60**. The transformations are in three places as shown in FIG. **5**:

**[0034]** A worker creation stage **62** may be used when local storage arises from OpenMP clauses. In an embodiment, the worker creation stage **62** replaces OpenMP directives with expansions for multiple workers. The worker creation stage **62** may also represent local storage using dynamic allocation. Pseudocode for the worker creation stage **62** is provided below.

```
Loop:
  b = alloca ...
  // the body of this loop references the local variable b
  ...
  <inner j-loop>
  ...
  End-loop:
```

**[0035]** A local storage expansion stage **64** handles a relatively large portion of the transformations described herein. In one example, the local storage expansion stage **64** handles allocation and referencing of private variables that remain. The pass of the illustrated stage **64** is conducted relatively late to allow other compiler optimizations to registerize local variables as far as possible. Accordingly, variables that could not otherwise be registerized are dealt with in the stage **64**. If a loop has a set S of private variables, then the stage **64** creates an array of type S with dimension the pipeline depth, which is dynamic count of iterations in flight.

**[0036]** A dataflow operation conversion stage **66** may handle the management of the individual slots in the private variable array created for each loop.

**[0037]** Worker Creation

**[0038]** The worker creation stage **62** may create multiple workers as directed by OpenMP directives. For non-OpenMP loops, the worker creation stage **62** may automatically decide the number of workers to generate. Similarly, OpenMP directives may specify the pipeline depth, or the compiler **60** may select the degree of pipelining to generate. For the purposes of discussion, it is assumed that two workers are created and that a pipeline depth of three is selected.

**[0039]** A pair of LLVM IR intrinsics may be introduced to support loop-local storage:

**[0040]** r=pipeline.limited.entry(int id, int depth)

**[0041]** pipeline.limited.exit(r)

**[0042]** These intrinsics enclose the loops that need local storage. The arguments of the "entry" call specify the pipeline depth and mark the place where allocation for the enclosed loops occurs. The "exit" marks the deallocation point. This representation ensures that independent of the number of workers generated, a single allocation/deallocation is done for the loops.

**[0043]** Pseudo-code of the original single loop after the worker creation stage **62** is shown below. In the illustrated example, the original loop has been replicated to form two workers. Additionally, the local variable in the original loop becomes a separate local variable in each of the new loops. Pipelining has not been accounted for yet and is done later in the local storage expansion stage **64**. The pseudo-code after processing by the worker creation stage **62** might be:

```
    depth.region = pipeline.limited.entry(id1, 3)
    // Worker 0
    region0 = parallel.region.entry(id2)
Loop0:
  b.priv.0 = alloca ...
  // all uses of b in the loop are replaced with b.priv.0
    ...
    <inner j-loop>
    ...
End-loop0:
    parallel.region.exit(region0)
    // Worker 1
    region1 = parallel.region.entry(id3)
Loop1:
  b.priv.1 = alloca ...
  // all uses of b in the loop are replaced with b.priv.1
    ...
    <inner j-loop>
    ...
End-loop1:
    parallel.region.exit(regionl)
    pipeline.limited.exit(depth.region)
```

**[0044]** Local Storage Expansion

**[0045]** In an embodiment, the local storage expansion stage **64** performs the transformation to account for pipelining. The pipeline depth of three is enforced using the concept of a token and a pool of three token values is created for each worker. In one example, an iteration may begin when a token can be obtained from the pool. This operation is modeled by a call to "token.take", which completes only when a local storage slot becomes available. When an iteration is completed, the token is returned to the pool. This return is modeled by a call to "token.return". In one example, since only three distinct token values exist, only three iterations can execute concurrently in each worker.

**[0046]** Pseudo-code after the local storage expansion stage **64** might be:

```
// Local variable pool declaration
#define num_workers 2
#define depth 3
struct worker_pool {
  struct loop_pool {
    double B[100];
  } ls[depth];
} pool[num_workers];
  // Allocate the pool
  pool = CsaMemAlloc(sizeof(worker_pool));
  // Worker 0
  w0 = &pool[0];
```

-continued

```
Loop0:
  // token.take will return one of these values:
  // &w0.1s[0], &w0.1s[1], ..., w0.1s[depth-1]
  w0_pool = token.take(w0, sizeof(ls), depth);
  B_loop_local = &w0_pool.B;
  // all uses of B in the loop are replaced with B_loop_local
  ...
  <inner j-loop>
  token.return(pool, w0_pool);
End-loop0:
  // Worker 1
  w1 = &pool[1];
Loop 1:
  // token.take will return one of these values:
  // &w1.1s[0], &w1.1s[1], ..., w1.1s[depth-1]
  w1_pool = token.take(w1, sizeof(ls), depth);
  B_loop_local = &w1_pool.B;
  // all uses of B in the loop are replaced with B_loop_local
  ...
  <inner j-loop>
  token.return(pool, w1_pool);
End-loop 1:
  // Deallocate the pool
  CsaMemFree(pool);
```

**[0047]** Dataflow Operation Conversion

**[0048]** The final stage in implementing loop-local storage is during the dataflow operation conversion stage **66**, which converts IR code into dataflow operations. The intrinsics token.take and token.return may be abstract representations of a mechanism that doles out a fixed number of tokens. In an embodiment, the physical implementation of this mechanism uses CSA LICs. The fundamental property of CSA LICs is to hold multiple values, to deliver values from one end of the LIC when read, and to write values at the other end of the LIC when written. This property may be used to permit only a fixed number of values to circulate through the loop body. In one example, the depth of the LIC is chosen to be the user-specified pipeline depth. Additionally, the values in the LIC may be offsets of individual slots allocated for the private variables of a loop. When a new iteration of the loop begins, a value is read from the LIC and added to a base address to generate the slot address for the current iteration of the loop. When the iteration completes, the offset may be written back to the LIC. Because the LIC holds only "depth" number of values, only depth number of iterations may execute concurrently, with each using a separate local storage slot. Example dataflow operations that implement this scheme are shown below.

**[0049]** In a dataflow machine, instructions execute when their input dependencies are satisfied. In the following, an "inord" is an input ordinal (e.g., a signal that an input dependence has been satisfied) and an "outord" is generated by an instruction when the instruction completes execution to indicate that the result in now available. The gate64, add64 and mov instructions are explained first, and then their use in implementing token.take and token.return.

**[0050]** gate64 result, inord, value

**[0051]** The instruction does not execute until inord is available. Then, "value" becomes available as the result.

**[0052]** add64 result, input1, input2

**[0053]** The instruction does not execute until input1 and input2 are available. Then, "result" becomes available as the sum of "input1" and "input2".

**[0054]** mov0 result, inord, value

**[0055]** The instruction does not execute until "inord" is available. Then, "value" becomes available as the result.

**[0056]** The pseudocode below is an example output of the dataflow operation conversion stage **66** for a CSA implementation.

```
// Each loop iteration requires 400 bytes of local storage
// There are 2 workers created for the original loop
// A pipeline depth of 3 is implemented
// Total local storage = 400 * 3 * 2 bytes = 2400 bytes
// Worker0 uses a pool that ranges from bytes 0 to 1199
// Worker1 uses a pool that ranges from bytes 1200 to 2399
// Within each worker's pool, the 3 slots have offsets 0, 400, 800
// A LIC of depth 3 is initialized with offset values:
// offset_of(slot0), offset_of(slot1), offset_of(slot2)
.lic@8 .i64 %slot_offset
...
% slot_offset:ci64 = init64 0
% slot_offset:ci64 = init64 400
% slot_offset:ci64 = init64 800
...
...
// token_take implemented on CSA
// Dynamic memory allocation outside the loop generates the pool address
pool = ...... // Equivalent of CsaMemAlloc(2400)
...
// In the loop, when the token_take is ready to execute
//the pool address is made available to the add64 instruction
gate64 pool_gated, token_take_inord, pool
// The address of the local storage slot assigned to this iteration
// is computed
add64 slot_addr,_slot_offsets,_pool_gated
....
....
// token_return implemented on CSA
// In the loop, when the token_return is ready to execute
// The slot_offset is written back at the end of the LIC
gate64 slot_offsets, token_return_inord, slot_offsets
// the completion of token_return is signaled with this mov0
mov0 token_return_outord, token_return inord
```

**[0057]** In this way, the dataflow properties of CSA LICs are exploited to enable pipelining of parallel loops while guaranteeing that enough local storage is available for dynamic loop iterations. The compiler **60** may conduct this transformation automatically and a prototype OpenMP language extension has been implemented to demonstrate the advantages of the solution.

**[0058]** Turning now to FIG. **6**, a performance-enhanced computing system **151** is shown. The system **151** may generally be part of an electronic device/platform having computing functionality (e.g., personal digital assistant/PDA, notebook computer, tablet computer, convertible tablet, server), communications functionality (e.g., smart phone), imaging functionality (e.g., camera, camcorder), media playing functionality (e.g., smart television/TV), wearable functionality (e.g., watch, eyewear, headwear, footwear, jewelry), vehicular functionality (e.g., car, truck, motorcycle), robotic functionality (e.g., autonomous robot), Internet of Things (IoT) functionality, etc., or any combination thereof. In the illustrated example, the system **151** includes a host processor **153** (e.g., central processing unit/CPU) having an integrated memory controller (IMC) **155** that is coupled to a system memory **157**.

**[0059]** The illustrated system **151** also includes an input output (**10**) module **159** implemented together with the host processor **153** and a graphics processor **161** (e.g., graphics processing unit/GPU) on a semiconductor die **163** as a system on chip (SoC). The illustrated IO module **159** communicates with, for example, a display **165** (e.g., touch screen, liquid crystal display/LCD, light emitting diode/LED display), a network controller **167** (e.g., wired and/or wireless), and mass storage **169** (e.g., hard disk drive/HDD, optical disk, solid state drive/SSD, flash memory).

**[0060]** In an embodiment, the host processor **153**, the graphics processor **161** and/or the IO module **159** execute instructions **171** retrieved from the system memory **157** and/or the mass storage **169** to perform one or more aspects of the method **50** (FIG. **4**), already discussed. Thus, execution of the illustrated instructions **171** may cause the computing system **151** to detect one or more local variables in source code, wherein the one or more local variables lack dependencies across iterations of a loop in the source code, automatically generate pipeline execution code for the one or more local variables, and incorporate the pipeline execution code into an output of a compiler.

**[0061]** In an embodiment, the pipeline execution code includes an initialization of a pool of buffer storage for the one or more local variables. In such a case, the initialized pool of buffer storage may be greater than (e.g., several multiples of) a local storage amount corresponding to a single iteration of the loop. Moreover, the pipelined execution code may further include a definition of a plurality of tokenized slots in the initialized pool of buffer storage, where each tokenized slot corresponds to a pipelined iteration of the loop. In an embodiment, the pipelined execution code further includes a pipeline depth definition. In one example, the local variable(s) are detected after a registerization of the source code and the source code is associated with a communication channel in a dataflow graph. Additionally, the automatic generation of the pipeline execution code may be conducted in response to the detection of the local variable(s).

**[0062]** The illustrated system **151** is therefore performance-enhanced at least to the extent that the pipelining of loops containing private data improves throughput. Indeed, the overall cycles needed to execute a loop may be significantly less than the product of static loop cycles and the loop iteration count.

**[0063]** FIG. **7** shows a semiconductor package apparatus **173**. The illustrated apparatus **173** includes one or more substrates **175** (e.g., silicon, sapphire, gallium arsenide) and logic **177** (e.g., transistor array and other integrated circuit/IC components) coupled to the substrate(s) **175**. The logic **177** may be implemented at least partly in configurable logic or fixed-functionality logic hardware. In one example, the logic **177** implements one or more aspects of the method **50** (FIG. **4**), already discussed. Thus, the logic **177** may detect one or more local variables in source code, wherein the local variable(s) lack dependencies across iterations of a loop in the source code, automatically generate pipeline execution code for the local variable(s), and incorporate the pipeline execution code into an output of a compiler. The illustrated apparatus **173** is therefore performance-enhanced at least to the extent that the pipelining of loops containing private data improves throughput. Indeed, the overall cycles needed to execute a loop may be significantly less than the product of static loop cycles and the loop iteration count.

**[0064]** In one example, the logic **177** includes transistor channel regions that are positioned (e.g., embedded) within the substrate(s) **175**. Thus, the interface between the logic **177** and the substrate(s) **175** may not be an abrupt junction. The logic **177** may also be considered to include an epitaxial layer that is grown on an initial wafer of the sub strate(s) **175**.

**[0065]** FIG. **8** illustrates a processor core **200** according to one embodiment. The processor core **200** may be the core for any type of processor, such as a micro-processor, an embedded processor, a digital signal processor (DSP), a network processor, or other device to execute code. Although only one processor core **200** is illustrated in FIG. **8**, a processing element may alternatively include more than one of the processor core **200** illustrated in FIG. **8**. The processor core **200** may be a single-threaded core or, for at least one embodiment, the processor core **200** may be multithreaded in that it may include more than one hardware thread context (or "logical processor") per core.

**[0066]** FIG. **8** also illustrates a memory **270** coupled to the processor core **200**. The memory **270** may be any of a wide variety of memories (including various layers of memory hierarchy) as are known or otherwise available to those of skill in the art. The memory **270** may include one or more code **213** instruction(s) to be executed by the processor core **200**, wherein the code **213** may implement one or more aspects of the method **50** (FIG. **4**), already discussed. The processor core **200** follows a program sequence of instructions indicated by the code **213**. Each instruction may enter a front end portion **210** and be processed by one or more decoders **220**. The decoder **220** may generate as its output a micro operation such as a fixed width micro operation in a predefined format, or may generate other instructions, microinstructions, or control signals which reflect the original code instruction. The illustrated front end portion **210** also includes register renaming logic **225** and scheduling logic **230**, which generally allocate resources and queue the operation corresponding to the convert instruction for execution.

**[0067]** The processor core **200** is shown including execution logic **250** having a set of execution units **255-1** through **255**-N. Some embodiments may include a number of execution units dedicated to specific functions or sets of functions. Other embodiments may include only one execution unit or one execution unit that can perform a particular function. The illustrated execution logic **250** performs the operations specified by code instructions.

**[0068]** After completion of execution of the operations specified by the code instructions, back end logic **260** retires the instructions of the code **213**. In one embodiment, the processor core **200** allows out of order execution but requires in order retirement of instructions. Retirement logic **265** may take a variety of forms as known to those of skill in the art (e.g., re-order buffers or the like). In this manner, the processor core **200** is transformed during execution of the code **213**, at least in terms of the output generated by the decoder, the hardware registers and tables utilized by the register renaming logic **225**, and any registers (not shown) modified by the execution logic **250**.

**[0069]** Although not illustrated in FIG. **8**, a processing element may include other elements on chip with the processor core **200**. For example, a processing element may include memory control logic along with the processor core **200**. The processing element may include I/O control logic and/or may include I/O control logic integrated with memory control logic. The processing element may also include one or more caches.

**[0070]** Referring now to FIG. **9**, shown is a block diagram of a computing system **1000** embodiment in accordance with an embodiment. Shown in FIG. **9** is a multiprocessor system **1000** that includes a first processing element **1070** and a second processing element **1080**. While two processing elements **1070** and **1080** are shown, it is to be understood that an embodiment of the system **1000** may also include only one such processing element.

**[0071]** The system **1000** is illustrated as a point-to-point interconnect system, wherein the first processing element **1070** and the second processing element **1080** are coupled via a point-to-point interconnect **1050**. It should be understood that any or all of the interconnects illustrated in FIG. **9** may be implemented as a multi-drop bus rather than point-to-point interconnect.

**[0072]** As shown in FIG. **9**, each of processing elements **1070** and **1080** may be multicore processors, including first and second processor cores (i.e., processor cores **1074***a* and **1074***b* and processor cores **1084***a* and **1084***b*). Such cores **1074***a*, **1074***b*, **1084***a*, **1084***b* may be configured to execute instruction code in a manner similar to that discussed above in connection with FIG. **8**.

**[0073]** Each processing element **1070**, **1080** may include at least one shared cache **1896***a*, **1896***b*. The shared cache **1896***a*, **1896***b* may store data (e.g., instructions) that are utilized by one or more components of the processor, such as the cores **1074***a*, **1074***b* and **1084***a*, **1084***b*, respectively. For example, the shared cache **1896***a*, **1896***b* may locally cache data stored in a memory **1032**, **1034** for faster access by components of the processor. In one or more embodiments, the shared cache **1896***a*, **1896***b* may include one or more mid-level caches, such as level 2 (L2), level 3 (L3), level 4 (L4), or other levels of cache, a last level cache (LLC), and/or combinations thereof.

**[0074]** While shown with only two processing elements **1070**, **1080**, it is to be understood that the scope of the embodiments is not so limited. In other embodiments, one or more additional processing elements may be present in a given processor. Alternatively, one or more of processing elements **1070**, **1080** may be an element other than a processor, such as an accelerator or a field programmable gate array. For example, additional processing element(s) may include additional processors(s) that are the same as a first processor **1070**, additional processor(s) that are heterogeneous or asymmetric to processor a first processor **1070**, accelerators (such as, e.g., graphics accelerators or digital signal processing (DSP) units), field programmable gate arrays, or any other processing element. There can be a variety of differences between the processing elements **1070**, **1080** in terms of a spectrum of metrics of merit including architectural, micro architectural, thermal, power consumption characteristics, and the like. These differences may effectively manifest themselves as asymmetry and heterogeneity amongst the processing elements **1070**, **1080**. For at least one embodiment, the various processing elements **1070**, **1080** may reside in the same die package.

**[0075]** The first processing element **1070** may further include memory controller logic (MC) **1072** and point-to-point (P-P) interfaces **1076** and **1078**. Similarly, the second processing element **1080** may include a MC **1082** and P-P interfaces **1086** and **1088**. As shown in FIG. **9**, MC's **1072** and **1082** couple the processors to respective memories, namely a memory **1032** and a memory **1034**, which may be portions of main memory locally attached to the respective processors. While the MC **1072** and **1082** is illustrated as integrated into the processing elements **1070**, **1080**, for alternative embodiments the MC logic may be discrete logic outside the processing elements **1070**, **1080** rather than integrated therein.

**[0076]** The first processing element **1070** and the second processing element **1080** may be coupled to an I/O subsystem **1090** via P-P interconnects **1076** **1086**, respectively. As shown in FIG. **9**, the I/O subsystem **1090** includes P-P interfaces **1094** and **1098**. Furthermore, I/O subsystem **1090** includes an interface **1092** to couple I/O subsystem **1090** with a high performance graphics engine **1038**. In one embodiment, bus **1049** may be used to couple the graphics engine **1038** to the I/O subsystem **1090**. Alternately, a point-to-point interconnect may couple these components.

**[0077]** In turn, I/O subsystem **1090** may be coupled to a first bus **1016** via an interface **1096**. In one embodiment, the first bus **1016** may be a Peripheral Component Interconnect (PCI) bus, or a bus such as a PCI Express bus or another third generation I/O interconnect bus, although the scope of the embodiments are not so limited.

**[0078]** As shown in FIG. **9**, various I/O devices **1014** (e.g., biometric scanners, speakers, cameras, sensors) may be coupled to the first bus **1016**, along with a bus bridge **1018** which may couple the first bus **1016** to a second bus **1020**. In one embodiment, the second bus **1020** may be a low pin count (LPC) bus. Various devices may be coupled to the second bus **1020** including, for example, a keyboard/mouse **1012**, communication device(s) **1026**, and a data storage unit **1019** such as a disk drive or other mass storage device which may include code **1030**, in one embodiment. The illustrated code **1030** may implement one or more aspects of the method **50** (FIG. **4**), already discussed. Further, an audio I/O **1024** may be coupled to second bus **1020** and a battery **1010** may supply power to the computing system **1000**.

**[0079]** Note that other embodiments are contemplated. For example, instead of the point-to-point architecture of FIG. **9**, a system may implement a multi-drop bus or another such communication topology. Also, the elements of FIG. **9** may alternatively be partitioned using more or fewer integrated chips than shown in FIG. **9**.

## Additional Notes and Examples

**[0080]** Example 1 includes a performance-enhanced computing system comprising a network controller, a processor coupled to the network controller, and a memory coupled to the processor, the memory including a set of executable program instructions, which when executed by the processor, cause the processor to detect one or more local variables in source code, wherein the one or more local variables lack dependencies across iterations of a loop in the source code, automatically generate pipeline execution code for the one or more local variables, and incorporate the pipeline execution code into an output of the compiler.

**[0081]** Example 2 includes the computing system of Example 1, wherein the pipeline execution code is to include an initialization of a pool of buffer storage for the one or more local variables.

**[0082]** Example 3 includes the computing system of Example 2, wherein the initialized pool of buffer storage is to be greater than a local storage amount corresponding to a single iteration of the loop.

**[0083]** Example 4 includes the computing system of Example 2, wherein the pipeline execution code is to further include a definition of a plurality of tokenized slots in the initialized pool of buffer storage, and wherein each tokenized slot is to correspond to a pipelined iteration of the loop.

**[0084]** Example 5 includes the computing system of Example 1, wherein the pipeline execution code is to include a pipeline depth definition.

**[0085]** Example 6 includes the computing system of any one of Examples 1 to 5, wherein the one or more local variables are to be detected after a registerization of the source code, automatic generation of the pipeline execution code is to be in response to detection of the one or more local variables, and the source code is to be associated with a communication channel in a dataflow graph.

**[0086]** Example 7 includes a semiconductor apparatus comprising one or more substrates, and logic coupled to the one or more substrates, wherein the logic is implemented at least partly in one or more of configurable logic or fixed-functionality hardware logic, the logic coupled to the one or more substrates to detect one or more local variables in source code, wherein the one or more local variables lack dependencies across iterations of a loop in the source code, automatically generate pipeline execution code for the one or more local variables, and incorporate the pipeline execution code into an output of a compiler.

**[0087]** Example 8 includes the semiconductor apparatus of Example 7, wherein the pipeline execution code is to include an initialization of a pool of buffer storage for the one or more local variables.

**[0088]** Example 9 includes the semiconductor apparatus of Example 8, wherein the initialized pool of buffer storage is to be greater than a local storage amount corresponding to a single iteration of the loop.

**[0089]** Example 10 includes the semiconductor apparatus of Example 8, wherein the pipeline execution code is to further include a definition of a plurality of tokenized slots in the initialized pool of buffer storage, and wherein each tokenized slot is to correspond to a pipelined iteration of the loop.

**[0090]** Example 11 includes the semiconductor apparatus of Example 7, wherein the pipeline execution code is to include a pipeline depth definition.

**[0091]** Example 12 includes the semiconductor apparatus of any one of Examples 7 to 11, wherein the one or more local variables are to be detected after a registerization of the source code, automatic generation of the pipeline execution code is to be in response to detection of the one or more local variables, and the source code is to be associated with a communication channel in a dataflow graph.

**[0092]** Example 13 includes the semiconductor apparatus of any one of Examples 7 to 12, wherein the logic coupled to the one or more substrates includes transistor channel regions that are positioned within the one or more substrates.

**[0093]** Example 14 includes at least one computer readable storage medium comprising a set of instructions, which when executed by a computing system, cause the computing system to detect one or more local variables in source code, wherein the one or more local variables lack dependencies across iterations of a loop in the source code, automatically generate pipeline execution code for the one or more local variables, and incorporate the pipeline execution code into an output of a compiler.

**[0094]** Example 15 includes the at least one computer readable storage medium of Example 14, wherein the pipeline execution code is to include an initialization of a pool of buffer storage for the one or more local variables.

**[0095]** Example 16 includes the at least one computer readable storage medium of Example 15, wherein the initialized pool of buffer storage is to be greater than a local storage amount corresponding to a single iteration of the loop.

**[0096]** Example 17 includes the at least one computer readable storage medium of Example 15, wherein the pipeline execution code is to further include a definition of a plurality of tokenized slots in the initialized pool of buffer storage, and wherein each tokenized slot is to correspond to a pipelined iteration of the loop.

**[0097]** Example 18 includes the at least one computer readable storage medium of Example 14, wherein the pipeline execution code is to include a pipeline depth definition.

**[0098]** Example 19 includes the at least one computer readable storage medium of any one of Examples 14 to 18, wherein the one or more local variables are to be detected after a registerization of the source code, automatic generation of the pipeline execution code is to be in response to detection of the one or more local variables, and the source code is to be associated with a communication channel in a dataflow graph.

**[0099]** Example 20 includes a method of operating a compiler, the method comprising detecting one or more local variables in source code, wherein the one or more local variables lack dependencies across iterations of a loop in the source code, automatically generating pipeline execution code for the one or more local variables, and incorporating the pipeline execution code into an output of the compiler.

**[0100]** Example 21 includes the method of Example 20, wherein the pipeline execution code includes an initialization of a pool of buffer storage for the one or more local variables.

**[0101]** Example 22 includes the method of Example 21, wherein the initialized pool of buffer storage is to be greater than a local storage amount corresponding to a single iteration of the loop.

**[0102]** Example 23 includes the method of Example 21, wherein the pipeline execution code further includes a definition of a plurality of tokenized slots in the initialized pool of buffer storage, and wherein each tokenized slot is to correspond to a pipelined iteration of the loop.

**[0103]** Example 24 includes the method of Example 20, wherein the pipeline execution code includes a pipeline depth definition.

**[0104]** Example 25 includes the method of any one of Examples 20 to 24, wherein the one or more local variables are detected after a registerization of the source code, automatic generation of the pipeline execution code is in response to detection of the one or more local variables, and the source code is associated with a communication channel in a dataflow graph.

**[0105]** Example 26 includes means for performing the method of any one of Examples 20 to 25.

**[0106]** Thus, technology described herein may include an automated compiler transformation that can take as input a loop that has some form of local loop storage and dynamically pipeline the loop using one or more workers for a dataflow architecture such as CSA. The compiler may detect local storage remaining in loops after registerization and allocate enough memory to hold the private variables for a) each worker, and b) each concurrent execution of a worker.

As each worker body commences execution, the worker body may be assigned a unique slot in the allocated private storage. When the worker completes execution of an iteration, the local storage slot associated with the worker may be automatically recycled for use in future iterations.

**[0107]** Several applications/benchmarks such as, for example, the SPGemm (sparse matrix-matrix multiplication) and Apriori benchmarks, may benefit from the transformation technology described herein.

**[0108]** Embodiments are applicable for use with all types of semiconductor integrated circuit ("IC") chips. Examples of these IC chips include but are not limited to processors, controllers, chipset components, programmable logic arrays (PLAs), memory chips, network chips, systems on chip (SoCs), SSD/NAND controller ASICs, and the like. In addition, in some of the drawings, signal conductor lines are represented with lines. Some may be different, to indicate more constituent signal paths, have a number label, to indicate a number of constituent signal paths, and/or have arrows at one or more ends, to indicate primary information flow direction. This, however, should not be construed in a limiting manner. Rather, such added detail may be used in connection with one or more exemplary embodiments to facilitate easier understanding of a circuit. Any represented signal lines, whether or not having additional information, may actually comprise one or more signals that may travel in multiple directions and may be implemented with any suitable type of signal scheme, e.g., digital or analog lines implemented with differential pairs, optical fiber lines, and/or single-ended lines.

**[0109]** Example sizes/models/values/ranges may have been given, although embodiments are not limited to the same. As manufacturing techniques (e.g., photolithography) mature over time, it is expected that devices of smaller size could be manufactured. In addition, well known power/ground connections to IC chips and other components may or may not be shown within the figures, for simplicity of illustration and discussion, and so as not to obscure certain aspects of the embodiments. Further, arrangements may be shown in block diagram form in order to avoid obscuring embodiments, and also in view of the fact that specifics with respect to implementation of such block diagram arrangements are highly dependent upon the computing system within which the embodiment is to be implemented, i.e., such specifics should be well within purview of one skilled in the art. Where specific details (e.g., circuits) are set forth in order to describe example embodiments, it should be apparent to one skilled in the art that embodiments can be practiced without, or with variation of, these specific details. The description is thus to be regarded as illustrative instead of limiting.

**[0110]** The term "coupled" may be used herein to refer to any type of relationship, direct or indirect, between the components in question, and may apply to electrical, mechanical, fluid, optical, electromagnetic, electromechanical or other connections. In addition, the terms "first", "second", etc. may be used herein only to facilitate discussion, and carry no particular temporal or chronological significance unless otherwise indicated.

**[0111]** As used in this application and in the claims, a list of items joined by the term "one or more of" may mean any combination of the listed terms. For example, the phrases "one or more of A, B or C" may mean A; B; C; A and B; A and C; B and C; or A, B and C.

**[0112]** Those skilled in the art will appreciate from the foregoing description that the broad techniques of the embodiments can be implemented in a variety of forms. Therefore, while the embodiments have been described in connection with particular examples thereof, the true scope of the embodiments should not be so limited since other modifications will become apparent to the skilled practitioner upon a study of the drawings, specification, and following claims.

We claim:

**1**. A computing system comprising:

a network controller;

a processor coupled to the network controller; and

a memory coupled to the processor, the memory including a set of executable program instructions, which when executed by the processor, cause the processor to:

detect one or more local variables in source code, wherein the one or more local variables lack dependencies across iterations of a loop in the source code,

automatically generate pipeline execution code for the one or more local variables, and

incorporate the pipeline execution code into an output of a compiler.

**2**. The computing system of claim **1**, wherein the pipeline execution code is to include an initialization of a pool of buffer storage for the one or more local variables.

**3**. The computing system of claim **2**, wherein the initialized pool of buffer storage is to be greater than a local storage amount corresponding to a single iteration of the loop.

**4**. The computing system of claim **2**, wherein the pipeline execution code is to further include a definition of a plurality of tokenized slots in the initialized pool of buffer storage, and wherein each tokenized slot is to correspond to a pipelined iteration of the loop.

**5**. The computing system of claim **1**, wherein the pipeline execution code is to include a pipeline depth definition.

**6**. The computing system of claim **1**, wherein the one or more local variables are to be detected after a registerization of the source code, automatic generation of the pipeline execution code is to be in response to detection of the one or more local variables, and the source code is to be associated with a communication channel in a dataflow graph.

**7**. A semiconductor apparatus comprising:

one or more substrates; and

logic coupled to the one or more substrates, wherein the logic is implemented at least partly in one or more of configurable logic or fixed-functionality hardware logic, the logic coupled to the one or more substrates to:

detect one or more local variables in source code, wherein the one or more local variables lack dependencies across iterations of a loop in the source code;

automatically generate pipeline execution code for the one or more local variables; and

incorporate the pipeline execution code into an output of a compiler.

**8**. The semiconductor apparatus of claim **7**, wherein the pipeline execution code is to include an initialization of a pool of buffer storage for the one or more local variables.

**9**. The semiconductor apparatus of claim **8**, wherein the initialized pool of buffer storage is to be greater than a local storage amount corresponding to a single iteration of the loop.

**10**. The semiconductor apparatus of claim **8**, wherein the pipeline execution code is to further include a definition of a plurality of tokenized slots in the initialized pool of buffer storage, and wherein each tokenized slot is to correspond to a pipelined iteration of the loop.

**11**. The semiconductor apparatus of claim **7**, wherein the pipeline execution code is to include a pipeline depth definition.

**12**. The semiconductor apparatus of claim **7**, wherein the one or more local variables are to be detected after a registerization of the source code, automatic generation of the pipeline execution code is to be in response to detection of the one or more local variables, and the source code is to be associated with a communication channel in a dataflow graph.

**13**. The semiconductor apparatus of claim **7**, wherein the logic coupled to the one or more substrates includes transistor channel regions that are positioned within the one or more substrates.

**14**. At least one computer readable storage medium comprising a set of instructions, which when executed by a computing system, cause the computing system to:

detect one or more local variables in source code, wherein the one or more local variables lack dependencies across iterations of a loop in the source code;

automatically generate pipeline execution code for the one or more local variables; and

incorporate the pipeline execution code into an output of a compiler.

**15**. The at least one computer readable storage medium of claim **14**, wherein the pipeline execution code is to include an initialization of a pool of buffer storage for the one or more local variables.

**16**. The at least one computer readable storage medium of claim **15**, wherein the initialized pool of buffer storage is to be greater than a local storage amount corresponding to a single iteration of the loop.

**17**. The at least one computer readable storage medium of claim **15**, wherein the pipeline execution code is to further include a definition of a plurality of tokenized slots in the initialized pool of buffer storage, and wherein each tokenized slot is to correspond to a pipelined iteration of the loop.

**18**. The at least one computer readable storage medium of claim **14**, wherein the pipeline execution code is to include a pipeline depth definition.

**19**. The at least one computer readable storage medium of claim **14**, wherein the one or more local variables are to be detected after a registerization of the source code, automatic generation of the pipeline execution code is to be in response to detection of the one or more local variables, and the source code is to be associated with a communication channel in a dataflow graph.

**20**. A method comprising:

detecting one or more local variables in source code, wherein the one or more local variables lack dependencies across iterations of a loop in the source code;

automatically generating pipeline execution code for the one or more local variables; and

incorporating the pipeline execution code into an output of a compiler.

**21**. The method of claim **20**, wherein the pipeline execution code includes an initialization of a pool of buffer storage for the one or more local variables.

**22**. The method of claim **21**, wherein the initialized pool of buffer storage is to be greater than a local storage amount corresponding to a single iteration of the loop.

**23**. The method of claim **21**, wherein the pipeline execution code further includes a definition of a plurality of tokenized slots in the initialized pool of buffer storage, and wherein each tokenized slot is to correspond to a pipelined iteration of the loop.

**24**. The method of claim **20**, wherein the pipeline execution code includes a pipeline depth definition.

**25**. The method of claim **20**, wherein the one or more local variables are detected after a registerization of the source code, automatic generation of the pipeline execution code is in response to detection of the one or more local variables, and the source code is associated with a communication channel in a dataflow graph.

* * * * *