Saving PAM4 Bus Energy with SMOREs: Sparse Multi-level Opportunistic Restricted Encodings

Mike O’Connor*, Donghyuk Lee*, Niladrish Chatterjee, Michael B. Sullivan, Stephen W. Keckler
{moconnor, donghyukl, nchatterjee, misullivan, skeckler}@nvidia.com
NVIDIA
Santa Clara, California, USA

Abstract—Pulse Amplitude Modulation (PAM) uses multiple voltage levels as different data symbols, transferring multiple bits of data simultaneously, thereby enabling higher communication bandwidth without increased operating frequencies. However, dividing the voltage into more symbols leads to a smaller voltage difference between adjacent symbols, making the interface more vulnerable to crosstalk and power noise. GDDR6X adopts four-level symbols (PAM4) with Maximum Transition Avoidance (MTA) coding, which reduces the effects of crosstalk. However, current coding approaches can consume excess energy and produce excess power noise. This paper introduces novel energy reduction techniques for PAM interfaces, specifically demonstrating them for GDDR6X PAM4. Inspired by prior work on conventional single-ended I/O interfaces, we leverage the unused idle periods in DRAM channels between data transmissions to apply longer but more energy-efficient codes. To maximize the energy savings, we build multiple sparse encoding schemes to fit different sized gaps in the DRAM traffic. These sparse encodings can provide energy reductions of up to 52% when transferring 4-bit data using a 3-symbol sequence. We evaluate these coding techniques using an NVIDIA RTX 3090 baseline, a recent GPU which uses GDDR6X with PAM4 signaling. Our evaluation shows the opportunity for large energy savings at the DRAM I/O interface (28.2% on average) over many HPC/DL applications with minimal performance degradation.

Keywords—PAM4; Memory Interfaces; Sparse Representation; Energy-efficient Design

I. INTRODUCTION

As processor performance increases, so does the need for higher bandwidth DRAM. In particular, highly parallel GPUs have been the leading consumers of the highest bandwidth DRAM devices. The energy required to access DRAM can be significant in these systems. To meet the high bandwidth demands of these GPU memory systems, the most recent GDDR6X DRAMs use multi-level signaling to transmit data between the host processor and the memory [18]. Rather than using two voltage levels to indicate a ‘1’ or ‘0’ value bit on the wire, four voltage levels are used to indicate two bit values (‘00’, ‘01’, ‘10’, or ‘11’). This multi-level signaling, known as four-level Pulse Amplitude Modulation (PAM4), enables GDDR6X DRAMs to support per pin data rates exceeding 20 Gbit/sec [19].

Due to the nature of the signaling on the PAM4 bus, sending each symbol requires a different amount of energy, similar to traditional two-level signaling on a terminated I/O bus. Many different data encoding schemes have been proposed to minimize energy on traditional two-level data buses. While some of these techniques can be naïvely adapted to PAM4 signaling, many of the underlying assumptions that shaped these techniques must be revisited. For instance, prior encoding techniques like MiLC [28] or Base+XOR [14] leverage similarity between adjacent data elements sent to/from a DRAM. Increasingly, however, whole memory encryption is being introduced into CPUs [12], [13], [24] and GPUs [25]. Whole-memory encryption breaks these prior energy-saving techniques, as it eliminates data similarity altogether. Therefore, a solution that works well on encrypted data is desirable.

The PAM4 signaling implemented on the GDDR6X interface only has 225 mV separating each of the four voltage levels. The high data rates and small gaps between data symbols make the interface sensitive to crosstalk and power noise. As a result, GDDR6X implements a constrained Maximum Transition Avoidance (MTA) [22] bus encoding. This encoding ensures that the minimum voltage and maximum voltage symbols are never transmitted back-to-back. An additional data pin (conventionally used for DBI in previous DRAM generations) is used to enable this MTA encoding. Encoding techniques like Data Bus Inversion (DBI) [29] can provide benefits on random, encrypted data, but they could cause violations of the MTA guarantee if applied on top of MTA encoding. Any low-energy coding solution will have to meet similar restrictions.

Low-energy bus encodings based on the use of sparse codes can meet both of these requirements. An arbitrary group of bits can be encoded into a sequence of symbols that is longer than the minimum strictly needed. As a result, a set of symbol sequences can be selected for the encoding that simultaneously consumes the lowest energy and meets the restrictions of MTA encoding. However, these longer, sparse codes require more bandwidth than the minimum symbol encoding. The “More is Less” (MiL) paper [28] considered opportunistically using sparse codes when DRAM bandwidth was not being fully utilized, and gaps on the bus allowed the longer codes to be used without significant performance degradation. The MiL paper showed the CPU-based DDR4 system only saw back-to-back requests 13% of the time, providing ample opportunities to use longer codes. The characteristics

* Both authors contributed equally to the paper.
of throughput-optimized processors like GPUs are different. GPU architectures are designed to allow thousands of memory requests to be outstanding to exploit all of the memory system bandwidth. Additionally, GPUs increasingly support fixed-function units like tensor cores and ray-tracing accelerators that increase the memory bandwidth demand for the workloads that use them [22]. We aim to develop a solution that operates well in such high-bandwidth GPU environments.

Inspired by the opportunistic use of sparse codes in the MiL work, we apply sparse codes to reduce energy on data transfers for PAM4 GPU-DRAM interfaces, when idle time on the bus allows. We show that bandwidth-intensive GPU workloads still often leave small gaps on the data bus that can be exploited by low-overhead sparse codes. These sparse codes are limited to use only a subset of the four possible signal levels so that they meet the MTA restrictions. We can then apply a constrained version of Data Bus Inversion which saves additional energy, without breaking the MTA guarantees. We describe a simple mechanism to dynamically determine when to apply the low energy encodings, which does not require any change to the commands or pins on the existing DRAM interface. We evaluate these Sparse Opportunistic Multi-level Restricted Encodings (SMOREs) in a target system based on the NVIDIA RTX 3090 [22] and show that there are large potential DRAM I/O energy savings (28.2% on average) on many high performance computing and deep learning applications with negligible performance degradation. Our key contributions follow.

- We describe several emerging challenges affecting DRAM interfaces. New signaling technologies, demand for fully encrypted data in DRAM, and applications that consistently consume most of the available bandwidth challenge many of the assumptions of prior low-energy encoding schemes.
- We develop low power bus encoding schemes for PAM4, subject to the MTA requirement, which have different configurations of input data length, output encoded symbol length, and number of utilized voltage levels. We describe underlying trade-offs for applying these codes.
- We propose a mechanism to select the encoding based on the observed gap between DRAM commands. This scheme requires no additional DRAM commands, additional DRAM signals, or sharing of metadata between the DRAM and the host processor.

II. BACKGROUND

The GDDR6X interface used as a baseline for this work uses a four-level Pulse Amplitude Modulation (PAM4) bus with Maximum Transition Avoidance (MTA) encoding.

A. Pulse Amplitude Modulation (PAM)

_Pulse Amplitude Modulation_ is a signaling technology that uses different voltage levels as data symbols [2], [10], [32], [35], [36]. Conventional I/O interfaces use symbols consisting of one of two voltage levels to transfer one bit of data during every unit interval data transfer time. A Unit Interval (UI) is defined as a time to transfer one symbol over a transmission line, the reciprocal of which is the symbol rate or baud. Figure 1A shows two unit intervals with two different symbols being sent over the wire. It is common for high-speed interfaces to connect each data wire to the voltage source through a termination resistor. Thus, one symbol is at the voltage level and another symbol is at a voltage level determined by the ratio between the termination resistance and on-resistance of the driver. Due to this termination scheme, the interface draws current and consumes more energy while transferring lower-voltage symbols than higher-voltage symbols.

To enable higher bandwidth with the same clock frequency, systems can introduce more levels in the PAM I/O interface. Figure 1B shows an example of a PAM4 approach with four different voltage levels which can encode two bits per symbol. This approach provides twice the bandwidth of traditional two-level signaling when operating at the same frequency. Figure 2 shows details of the GDDR6X PAM4 implementation [11]. The different voltage levels are generated by combining three output drivers. To deliver a lower voltage symbol, more drivers are enabled inducing a higher current flow. The _resistance_ column in the table of Figure 2 shows the pull-up and pull-down resistances of the equivalent voltage-divider circuit, which produces the voltages shown in the next column. The different symbols require the different current (and corresponding energy) consumption shown in Figure 1B. In the diagram, we label the voltage level with the lowest energy consumption _L_0 and the one with the highest energy consumption _L_3.

However, using PAM4 signaling introduces some practical challenges, as PAM4 signaling is more vulnerable to power noise and crosstalk from activity in adjacent wires. With a 1.35V supply, the difference between the PAM4 symbols is only 225 mV. The reduced voltage difference between adjacent symbols, relative to two-level signaling, means that smaller variations in the signal voltage can result in the wrong symbol being detected. These problems are most severe when symbols are transitioning between the maximum and minimum voltage levels, _L_0 and _L_3, or vice versa. These maximum transition swings induce the most noise in neighboring signals. The maximum transition swings also require the most rapid change in voltage, producing the most vulnerable periods for crosstalk.
B. Maximum Transition Avoidance (MTA) Encoding

To address the issues caused by transitions between the highest and lowest voltage symbols, GDDR6X adopts Maximum Transition Avoidance (MTA) encoding [22], [30]. This technique encodes the data such that no 3ΔV transitions from \( L_0 \) to \( L_3 \) or \( L_3 \) to \( L_0 \) ever occur on the data bus. The MTA encoding is based on the 139 possible 4-symbol sequences that start with \( L_0 \), \( L_1 \), or \( L_2 \) and do not contain any 3ΔV transitions. Of these 139 sequences, the highest energy 11 sequences are discarded leaving a set of 128 symbols. In contrast, discarding the lowest energy 11 sequences would increase average energy by 2%. Table I shows an example MTA encoding. The MTA encoding process takes each 8-bit sequence that would be sent on a given wire, and breaks it into its most-significant bit, and the remaining 7 bits as shown in Figure 3. The 7-bit data is converted to one of the 128 4-symbol sequences via a synthesized lookup table. This encoding is performed for groups of eight signals. The remaining unencoded bit from each of the 8 data signals are combined to form a 4-symbol PAM4 sequence that is sent in parallel with the encoded data on a ninth data signal. Using this additional signal, no extra time is required to send the MTA encoded data. As one DBI signal is coupled with eight data wires in GDDR6X, which is designed to also be backwards compatible with earlier GDDR6, the DBI signal is used as this ninth data signal. Since the data sent on this DBI wire is not MTA encoded, a specialized design rule is recommended only for the DBI wire, for example, applying extra distance between DBI wire and data wires or adding extra ground shielding wires.

MTA encoding has one additional complexity to prevent 3ΔV transitions between successive encoded 4-symbol sequences. While each of the 128 encoded sequences starts with an \( L_0 \), \( L_1 \), or \( L_2 \) symbol, a sequence can end with an \( L_3 \) symbol. As a result, an \( L_3 \) to \( L_0 \) transition could occur from one sequence to the next. To prevent this situation, whenever a symbol sequence ends with an \( L_2 \) or \( L_3 \) symbol, the next encoded symbol on that wire will be sent inverted. An \( L_0 \) symbol will be transmitted as \( L_3 \), \( L_1 \) as \( L_2 \), \( L_2 \) as \( L_1 \), and \( L_3 \) as \( L_0 \). This inversion prevents the problematic 3ΔV transition between successive symbols. Special handling also occurs when there is no data to send after a data burst. GDDR6X has a one-command clock “postamble” on an idle bus after a data burst. This postamble sets the bus at the \( L_3 \) level. It also occurs when there is no data to send after a data burst.

III. Prior Approaches to Reducing I/O Energy

Many energy reduction technologies for traditional two-level DRAM interfaces have been proposed, which could be adapted to a PAM4 interface. Data Bus Inversion (DBI) is a popular technique to reduce energy used on many DRAMs today. The key idea behind this mechanism is to transfer a group of bits in either its original form or its inverted form. One additional bit of metadata is transferred to indicate the polarity of the data. The polarity is selected so as to minimize the energy, for example by inverting the data if more than half of the bits are 1.

An intuitive approach to apply DBI on a PAM4 I/O interface is to divide the two bits within a symbol into one MSB and one LSB and apply DBI to a group of MSBs and a group of LSBs, independently. In the example data mapping in

### Table I: Example 7 bit to 4 symbol MTA encoding.

<table>
<thead>
<tr>
<th>bits[2:0]</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>0001</td>
<td>0110</td>
<td>0111</td>
<td>0100</td>
<td>0101</td>
<td>0110</td>
<td>0111</td>
</tr>
<tr>
<td>0001</td>
<td>2000</td>
<td>0002</td>
<td>0202</td>
<td>0200</td>
<td>0202</td>
<td>0200</td>
<td>0202</td>
<td>0200</td>
</tr>
<tr>
<td>0010</td>
<td>1000</td>
<td>0101</td>
<td>1011</td>
<td>1011</td>
<td>1101</td>
<td>1101</td>
<td>1101</td>
<td>1101</td>
</tr>
<tr>
<td>0011</td>
<td>2200</td>
<td>0201</td>
<td>2101</td>
<td>2101</td>
<td>2011</td>
<td>2011</td>
<td>2011</td>
<td>2011</td>
</tr>
<tr>
<td>0100</td>
<td>2020</td>
<td>0202</td>
<td>0012</td>
<td>0013</td>
<td>0221</td>
<td>0221</td>
<td>0221</td>
<td>0221</td>
</tr>
<tr>
<td>0101</td>
<td>0021</td>
<td>0102</td>
<td>0102</td>
<td>0102</td>
<td>0212</td>
<td>0212</td>
<td>0212</td>
<td>0212</td>
</tr>
<tr>
<td>0110</td>
<td>1021</td>
<td>0122</td>
<td>0212</td>
<td>1113</td>
<td>0221</td>
<td>1131</td>
<td>1311</td>
<td>1311</td>
</tr>
<tr>
<td>0111</td>
<td>0113</td>
<td>0223</td>
<td>0232</td>
<td>2113</td>
<td>2131</td>
<td>2131</td>
<td>2131</td>
<td>2131</td>
</tr>
<tr>
<td>1000</td>
<td>0210</td>
<td>1021</td>
<td>0202</td>
<td>0112</td>
<td>0201</td>
<td>1210</td>
<td>1210</td>
<td>1210</td>
</tr>
<tr>
<td>1001</td>
<td>1102</td>
<td>2101</td>
<td>2021</td>
<td>2102</td>
<td>2202</td>
<td>2202</td>
<td>2202</td>
<td>2202</td>
</tr>
<tr>
<td>1010</td>
<td>1120</td>
<td>0131</td>
<td>1013</td>
<td>1122</td>
<td>1310</td>
<td>1212</td>
<td>1212</td>
<td>1212</td>
</tr>
<tr>
<td>1011</td>
<td>1023</td>
<td>1123</td>
<td>1132</td>
<td>1132</td>
<td>1313</td>
<td>1313</td>
<td>1313</td>
<td>1313</td>
</tr>
<tr>
<td>1100</td>
<td>1201</td>
<td>0123</td>
<td>0132</td>
<td>2112</td>
<td>2121</td>
<td>2121</td>
<td>2121</td>
<td>2121</td>
</tr>
<tr>
<td>1101</td>
<td>1320</td>
<td>1231</td>
<td>1312</td>
<td>1322</td>
<td>1321</td>
<td>2123</td>
<td>2123</td>
<td>2123</td>
</tr>
<tr>
<td>1110</td>
<td>1210</td>
<td>1121</td>
<td>1121</td>
<td>2112</td>
<td>2121</td>
<td>2121</td>
<td>2121</td>
<td>2121</td>
</tr>
<tr>
<td>1111</td>
<td>0231</td>
<td>2013</td>
<td>2310</td>
<td>1332</td>
<td>0133</td>
<td>2331</td>
<td>0233</td>
<td>2320</td>
</tr>
</tbody>
</table>
Whole-memory encryption works by providing full-bandwidth encryption to and from DRAM, avoiding cold-boot attacks and passive DRAM bus monitoring with minimal performance degradation. Whole-memory encryption is supported in current and upcoming AMD EPYC [13], Intel Xeon (Ice Lake) [24], and IBM POWER10 [12] CPUs, and full-bandwidth encryption is currently supported on AMD GPUs [25]. Whole-memory encryption adds to the DRAM I/O energy challenge by vastly increasing the data entropy and precluding the use of prior data-similarity-based energy reduction techniques such as MiL [28], Base+XOR [14], or others [15, 27].

Opportunistic Encoding Techniques for Energy Efficiency. Song and Ipek [28] propose an opportunistic encoding mechanism which translates data into energy-efficient but longer codes only when there are idle periods on the DRAM interface. The mechanism, “More is Less” (MiL), operates by waiting for the memory controller to detect that the bus will likely be idle after a given command is issued. When this situation is detected, the read or write command is sent with an indication that a longer, more efficient encoding will be used for the data. Thus, the technique uses what would likely be idle cycles on the interface to send the data in a more energy-efficient manner.

MiL proposed two encoding mechanisms. First is a 3-LWC code, which is a variant of a Limited-weight code (LWC), which translates two 4-bit data into a 15-bit 1-LWC code that presents data as the relative location of one “1” valued bit. It then merges two 1-LWC codes by ORing them and adding two bits of metadata. By carefully choosing the metadata, at most three “1” valued bits in the resulting 17-bits of data will be set (hence, a 3-LWC). Thus, the 3-LWC code requires slightly more than 2× the bandwidth of the unencoded 8-bit value. The other coding scheme, MiLC, is a sophisticated combination of DBI and a data-similarity exploiting code which chooses the most energy efficient of four options that are generated by combining XOR and DBI. The energy reduction in this coding is data dependent.

The MiL work was based on a DDR4 CPU memory system and was able to find many long gaps on the DRAM bus to apply the energy-efficient sparse codes. Emerging high-performance computing (HPC) and deep learning (DL) applications demand high compute and memory bandwidths [19]. To improve performance on these applications, some systems adopt domain-specific accelerators, which are often specialized for efficient memory accesses by either placing them near memory for shorter round-trip DRAM access latencies or by accessing DRAM in more organized way. These trends put further pressure on DRAM bandwidth. Without significant innovation in DRAM device and I/O technology, the limited bandwidth and energy efficiency of DRAM will likely be a major bottleneck in future high-performance computing systems [23].

Given the need to i) avoid maximum 3ΔV transitions and ii) operate on encrypted data, applying MiL’s approach of employing longer energy-efficient codes to PAM4 is attractive. Achieving energy and noise reduction requires codes short enough to exploit small gaps on a highly utilized memory bus while also eliminating use of 3ΔV transitions.

IV. Proposal

Our goal is to create energy-efficient sparse encodings using multi-level symbols that are restricted to avoid 3ΔV transitions that can be opportunistically sent during otherwise
idle gaps on the bus. In this section, we develop these Sparse Multi-level Opportunistic Restricted Encodings (SMOREs). We first characterize the GPU memory traffic for gaps we can exploit. Next, we evaluate various coding strategies for the PAM4 interface while maintaining the restriction that no $3\Delta V$ transitions can occur on a wire. Finally, we describe the mechanism to use these energy efficient encodings without any performance loss.

### A. Finding Gaps in GPU Memory Traffic

To understand the opportunity to exploit idle periods on the data bus, we first consider how read transactions are serviced in GDDR6X. Figure 4a shows several scenarios in which two read commands are serviced by the DRAM. The first example in Figure 4a shows two READ commands providing the commands and addresses at clock cycles $T_0$ and $T_2$. Upon receiving the first read request at cycle $T_0$, the DRAM decodes the command/address and brings the requested data from the corresponding cell array to the I/O interface. This takes a predefined amount of time, RL, the Read Latency. Starting at cycle $T_{RL}$, the DRAM begins to transfer the 256-bit read data using PAM4 over the data pins. The data is sent as 8 symbols on each of 16 data pins (plus two DBI pins). The data is sent on both edges of a clock that cycles at twice the rate of the command clock. Thus, the entire read response is sent in the two cycles from $T_{RL}$ to $T_{RL+2}$. The read command received at $T_2$ transmits its data in the two subsequent cycles $T_{RL+2}$ to $T_{RL+4}$.

If there are more than two clocks between successive READ commands, the additional gap will result in idle cycles on the data bus. For example, Figure 4b shows two READ commands separated by four clocks, being sent at $T_0$ and $T_4$. This situation leads to a two-clock idle period on the data bus (highlighted in gray). After finishing the first data transfer at $T_{RL+2}$, there is one command clock postamble where the L1 symbol is held on all the data bits. This postamble prevents any $3\Delta V$ transitions from the final symbol in a burst to an idle bus. After the postamble, the bus reverts to the lowest-energy L0 symbol at $T_{RL+3}$.

Our goal is to use gaps on the bus to transmit more energy-efficient encodings on the DRAM data bus. Figures 4c and 4d illustrate how we might use the different proposed SMOREs in different scenarios. To measure the potential opportunity to utilize these gaps on the data bus, we profiled the DRAM access pattern of various applications using an

---

**Fig. 4:** Applying energy efficient sparse encoding to GDDR6X PAM4 interface.
TABLE II: Configuration of an evaluated system [22].

<table>
<thead>
<tr>
<th>Component</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Units</td>
<td>82 stream multi processors</td>
</tr>
<tr>
<td>Last-Level Cache</td>
<td>6 MB in total</td>
</tr>
<tr>
<td>Memory System</td>
<td>384 bit total bus, 24 GB GDDR6X</td>
</tr>
<tr>
<td></td>
<td>936.2 Gbps total channel bandwidth</td>
</tr>
<tr>
<td></td>
<td>4 32-byte sectors per cacheline</td>
</tr>
<tr>
<td>GDDR6X</td>
<td>Data Rate (per pin): 19.5 Gbps</td>
</tr>
<tr>
<td></td>
<td>Power Supply: ( V_{DD} / V_{DDQ} = 1.35 ) V</td>
</tr>
<tr>
<td></td>
<td>Output Driver: ( R_{output} / R_{output} = 120/120 \Omega )</td>
</tr>
<tr>
<td></td>
<td>Termination: ( R_T = 40 \Omega )</td>
</tr>
</tbody>
</table>

NVIDIA GPU instrumentation tool and a GPU simulator that integrates graphics and compute pipelines, cache hierarchy, and memory system [34]. The configuration of the evaluated system is based on an NVIDIA RTX 3090 and is summarized in Table II. We evaluate 42 memory-intensive CUDA applications from the Rodinia [6] and Lonestar [4] benchmark suites, MLPerf [16], and Exascale workloads [1], [7], [9], [21], [26], [33] that include CoMD, HPGMG, lulesh, MCB, MiniAMR, and Nekbone.

Figure 5a shows the data channel idle cycles after a READ request. On average, 59.2% of consecutive READ requests are served without any idle period between them, 29.1% of them are serviced with a one clock idle period, and 6.9% of the requests have more than 16 idle cycles afterwards. We find similar trends in consecutive WRITE requests in Figure 5b which show similar gapless data transfers (59.1% of all WRITEs) and similar one clock idle period between consecutive WRITEs (30.2%).

This data shows that DRAM activity is usually quite intense. There may be long gaps between transactions, leading to a lower average bus utilization, but when DRAM activity is taking place, it typically occurs in high-bandwidth back-to-back bursts or almost-back-to-back transactions.

This data shows that, by far, the most common gap on the bus that can be used for more energy efficient transfers is one clock long. These one-cycle gaps are often caused by the memory controller issuing an activate command rather than a read or write command in a given cycle. The memory controller generally prioritizes issuing bank activate commands to maximize the available bank-level parallelism in order to sustain the bandwidth demands of the application.

The predominance of short, one-cycle gaps limits any potential energy-saving code to at most a 50% overhead. The MiL work leverages 3-LWC encoding that encodes 8 bits with 17 bits, a 2.125× longer code in conventional two-level signaling. Applying 3-LWC directly, using only two levels of the PAM4 signaling to ensure no more than three L1 symbols would be sent for any 8 bits of data would require 4.25× the transfer time than using PAM4. We will explore shorter codes able to exploit the short gaps we see on the GDDR6X bus and able to provide energy savings.

B. Energy Efficient Sparse Coding

The key idea of generating a sparse code is selecting the lowest energy symbols in longer codes. In PAM4 signaling, four different voltage levels are used to encode two bits of data. If we have two symbols that have 16 different symbol sequences (a 4-bit capacity) but transfer only 2 bits of data, we can choose the four lowest energy symbol sequences, which are L0L0, L0L1, L1L0, and L2L0. Since the current difference between L0 and L1 (9.4 mA) is much larger than one between L1 and L2 (\( I_{L1} - I_{L2} = 5.6 \) mA), using L2L0 is more energy efficient than using L1L1. The average PAM4 symbol requires 1057.5 fJ to send two bits of information (528.8 fJ/bit), but it requires only an average of 865 fJ to send two bits using the lower-energy set of two PAM4 symbols with the simple 2-bit to 2-symbol code above (432.5 fJ/bit). This code reduces the energy required to transmit two bits of information by 18% while requiring twice the time. We explored a range of alternative potential codes based the following parameters.

Input code length (bits). The number of data bits that are encoded at once is a key parameter. Encoding more bits simultaneously enables more energy efficient codes. However, the overhead of the lookup table (which is converted into synthesized logic gates) increases significantly with longer input code lengths. In this work, we choose coding schemes that encode 4 bits together, a sweet spot that provides the bulk of the benefits of longer input codes, while requiring relatively small area and energy overhead for the encoder. Furthermore, we wanted to keep the delay of the encoder less than or equal to that of MTA.

Output code length (symbols). The next parameter is the resulting code length after encoding, expressed as the number of symbols that will be sent on the bus. For example, the output code length of four symbols means that it will be transferred over four PAM4 symbols. The output code length determines the bandwidth overhead of the code. A longer sparse code can enable better energy efficiency but can be applied less often, as fewer cases have a long enough gap on the bus to use the code without introducing a performance penalty.

Utilized level count. The last parameter is how many voltage levels out of four in PAM4 are utilized for the output sparse code. If the output sparse code uses only the L0 and L1 levels, we call this a 2-level encoding in PAM4. Since our goal is reducing energy consumption, \( N \)-level encoding means that the sparse code uses the \( N \) smallest energy symbols (from L0 to L3). In this study, we use either 2-level or 3-level encoding for output sparse codes, which automatically limits the maximum transition to be 1\( \Delta V \) or 2\( \Delta V \), respectively. Since the maximum transition of PAM4 (3\( \Delta V \)) is fundamentally avoided, our sparse encoding schemes do not require any transition limiting mechanism on top of them like MTA. Allowing all four voltage levels in the sparse codes while also enforcing that no codes had 3\( \Delta V \) transitions resulted in no codes that contained any L3 symbols. Thus, there were no 4-level sparse codes to consider four bits of input.
Utilized level count is three, the number of different symbol examples, if the output code length is four symbols and the energy sequences from the 81 possible output combinations.

To further reduce energy consumption, we apply DBI on top of the sparse codes that use two or three voltage levels, the requirement of MTA to PAM4 is not possible. However, in our restricted space domain. Therefore, these two mechanisms are orthogonal and can be applied together if one mechanism does not break the assumptions of the other.

We first apply a sparse encoding, then apply a simple DBI-like mechanism. To apply DBI on a 3 level sparse code, we employ a simple level-swap scheme. The goal is to simply...
ensure that if any symbol is represented on the majority of the wires, it is swapped with the minimum-energy symbol, L0. The DBI signal then specifies whether we swapped L1 or L2 (or neither) with L0. This mechanism works for 2-level or 3-level codes.

- Swap L0 and L1 and set \( DBI = L1 \), if \( N_{L1} > 4 \)
- Swap L0 and L2 and set \( DBI = L2 \), if \( N_{L2} > 4 \)
- \( N_{L1} \) = the number of L1 levels in eight data wires.
- \( N_{L2} \) = the number of L2 levels in eight data wires.
- In the default case, \( DBI = L0 \)

**Level Shifting to Avoid Max Transitions between MTA and Sparse Codes.** While a 2-level or 3-level code does not have \( 3\Delta V \) transitions among its own symbols, a problem can occur if the immediately preceding MTA transaction ended with an L3 symbol. A code that starts with an L0 symbol would cause a max transition violation. The MTA code inverts the entire next encoded symbol sequence if the previous symbol ended on an L3. This choice does not significantly affect the energy-efficiency of the fairly dense 7-bit to 4-symbol MTA code. However, this approach is very bad for sparse codes, where a large number of minimum-energy L0 symbols would be converted to expensive L3 symbols.

Rather than inverting the code, the sparse codes employ a level shifting policy. If the previous symbol on the bus is an L3, then the next symbol is transmitted at a level one higher than it otherwise would have been. Thus, if an L3 symbol would have been followed by a problematic L0 symbol, it instead uses a level-shifted L1 symbol. If the next symbol is an L2, then the symbol is level shifted to an L3. In this latter case, the subsequent symbol will also be subject to level shifting. Since none of the codes considered start with L2L2, the level shifting affects at most two successive symbols.

The level shifting takes place after DBI has been applied. The receiver simply subtracts one level from any symbol received after an L3 symbol, and then applies the DBI-specified level swap if necessary. This simple approach is effective at avoiding any \( 3\Delta V \) swings on the seams between an MTA encoded burst and sparse-coded burst.

**Survey of Codes.** Figure 6 shows the average energy consumption (fJ/bit on the Y-axis) for sparse encoding schemes that translate 4-bit input codes to different output code lengths (shown as symbols on the X-axis). We first evaluate the energy consumption of baseline PAM4 signaling in different cases. In our estimation, the basic PAM4 signaling without MTA consumes 528.8 fJ/bit on average. Applying MTA to the basic PAM4 signaling consumes 8.7% more energy (574.8 fJ/bit on average), due mainly to the higher chance of using L1 symbols instead of L0 to avoid \( 3\Delta V \) transitions.

Another consideration for the baseline PAM4 signaling is whether it requires a postamble. As shown in Figure 6, if there is a gap on the data bus, a postamble of L1 level follows a data transfer for one clock period. Driving the postamble effectively adds an additional 325.4 fJ/bit to the burst energy, resulting in 900.2 fJ/bit (a 56.6% increase) for a burst followed by a gap. Since the number of transfers followed by gaps is application dependent, we show both PAM4 with MTA signaling both with and without postamble power in Figure 6.

The first observation is that using longer output code lengths leads to more potential energy savings. Second, 3-level encoding schemes consume lower energy than 2-level encoding schemes at the same output code length. This result is intuitive because, at the same output code length, the potential code space of the 3-level encoding is much larger and also includes all of the code space of the 2-level encoding. Therefore, a 3-level encoding can always provide an equal or better choice than 2-level encoding. The energy consumption difference between 2-level and 3-level encoding reduces with longer output code lengths. The code space increases while the 4-bit input always requires only 16 different symbol sequences. Therefore, choosing the 16 lowest energy symbol sequences in the longer output code sequences increasingly includes fewer of the higher energy L2 symbols. Eventually the two schemes converge on the same encodings.

Applying DBI provides additional energy savings for both 2-level encoding and 3-level encoding, but the degree of savings is reduced with longer output code lengths. Codes with DBI provide better energy efficiency than those without when there are conditions to invert data pins. In sparser output codes, the probability of more than four symbols on the bus simultaneously being non-zero diminishes.

Table IV shows the average energy for several encoding schemes, all of which avoid the maximum \( 3\Delta V \) transition except for the basic PAM4 (2-bit 1-symbol PAM4). Compared to two effective baselines, MTA enabled PAM4 signaling without/with postamble, our preferred, most useful sparse encoding (4-bit to 3 symbols (3-level) encoding with DBI), shows a 25%–52% reduction in per bit energy consumption.

**C. Mechanism**

To enable energy efficient encoding for READ requests, the DRAM must first detect the gap between DRAM requests, and then encode the requested data with a sparse code of appropriate code length. Upon receiving data, the GPU needs...
TABLE IV: Energy consumption of various encodings.

<table>
<thead>
<tr>
<th>Code</th>
<th>Energy (fJ/bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-bit 1 symbol PAM4</td>
<td>528.8</td>
</tr>
<tr>
<td>2-bit 1 symbol PAM4 w/ DBI</td>
<td>446.5</td>
</tr>
<tr>
<td>7-bit 4 symbols PAM4 (MTA)</td>
<td>574.8</td>
</tr>
<tr>
<td>7-bit 4 symbols PAM4 (MTA/postamble)</td>
<td>900.2</td>
</tr>
<tr>
<td>4-bit 3 symbols (3 level)</td>
<td>448.4</td>
</tr>
<tr>
<td>4-bit 3 symbols (3 level/DBI)</td>
<td>432.3</td>
</tr>
<tr>
<td>4-bit 4 symbols (3 level)</td>
<td>382.5</td>
</tr>
<tr>
<td>4-bit 4 symbols (3 level/DBI)</td>
<td>374.8</td>
</tr>
<tr>
<td>4-bit 6 symbols (3 level)</td>
<td>331.8</td>
</tr>
<tr>
<td>4-bit 6 symbols (3 level/DBI)</td>
<td>331.4</td>
</tr>
<tr>
<td>4-bit 8 symbols (3 level)</td>
<td>319.8</td>
</tr>
<tr>
<td>4-bit 8 symbols (3 level/DBI)</td>
<td>319.7</td>
</tr>
</tbody>
</table>

to know the expected length of the code and decode it with the appropriate decoder.

Detecting gap in DRAM. The MiL work had the host memory controller anticipate gaps between read commands and specify different read command types based on the encoding to be used. This approach requires additional command types and an additional command signal in GDDR6X. To minimize the cost, we propose a DRAM-based determination of which coding to use. As discussed in Section IV-A, there is a latency between the time of issuing READ/WRITE requests to the DRAM and the time the requested data is returned (or sent, in the case of writes). Since no public data exists with the exact READ/WRITE latency in GDDR6X, we used available values from similar GDDR5 DRAMs. The read latency (RL) is about 12 ns, which is about 30 clocks in our system configuration, and the write latency is about 5 cycles [5] which is much larger than the gaps we need to detect. Therefore, determining the gap at the time of receiving the next READ/WRITE request is early enough to encode the first response without any performance penalty.

Upon receiving a READ/WRITE request, the DRAM resets a counter that tracks the cycle difference between consecutive READ/WRITE requests. Upon receiving another READ/WRITE request, the DRAM determines the number of idle cycles and selects a sparse encoding scheme that will result in a code that fits into the available time on the bus. The major benefit of this mechanism is that it does not require any extra commands or extra pins.

Encoding in DRAM. After bringing data from DRAM cells to the I/O drivers, the DRAM encodes the data using the appropriate MTA or sparse code based on the detected gap (if any), which takes some amount of time. However, since our profiling results show that the dominant data bus gap is one clock, we expect that using one sparse encoding scheme for a one clock gap for all gapped accesses can provide most of benefits of a more sophisticated dynamic encoding scheme. Based on these, we choose to use 4-bit to 3 symbols (3-level), with the exact timing shown in Figure 4d.

In our baseline system without any sparse encoding, DRAM must apply an MTA encoding. Since our sparse encodings always avoid the maximum 3ΔV transition, DRAM could apply MTA for gapless requests and a sparse encoding for consecutive requests with idle channel periods. Based on our evaluation Section V, applying 4-bit to 3 symbols (3-level) encoding requires a small latency similar to that MTA. Therefore, we expect our mechanism can be applied without any penalty over the baseline DRAM timings.

Decoding in the GPU. When issuing successive READ/WRITE requests, the GPU tracks the cycle counts between the requests. The gaps between read requests allow the GPU to anticipate the encoding format the DRAM will use to return the data. The GPU then decodes received data either with the appropriate MTA decoder or with a sparse decoder. Again, since the MTA decoder takes similar time as the sparse decoder, we expect no additional bandwidth or latency penalty for this decoding scheme.

Sparse encoding can start immediately when the data is available and stretches over the required time to send the sparse code. However, sparse decoding must generally be postponed until the entire sparse code has arrived. Since our preferred mechanism only extends the sparse burst by a single cycle, the performance penalty is negligible, 0.024% on average (0.15% worst case), over the benchmarks evaluated.

While we have described the encoding and decoding for reads, handling write data is similar. The GPU encodes write data before sending it to the I/O drivers, the DRAM encodes the data using the appropriate MTA or sparse code based on the detected gap, and the DRAM decodes the data based on the gaps it sees between the write commands.

V. Evaluation

In this section, we evaluate the benefits of our proposed SMOREs mechanism to reduce I/O energy consumption in the GDDR6X PAM4 interface. Towards this end, we use a GPU simulator that models the detailed DRAM-based memory system, using GDDR6X parameters estimated from public information and the specifications of the Graphics SDRAM family of products, e.g., GDDR5, GDDR6, etc. [3], [5], [17], [18], [20]. We evaluate a system with an NVIDIA RTX 3090 [22] GPU, the most recent GPU with GDDR6X. The system configuration is listed in Table I.

We evaluate CUDA applications from the Rodinia [6] and Lonestar [4] benchmark suites, MLPerf [16], and Exascale workloads [1], [7], [9], [21], [26], [33]. These are the same applications that used for profiling idle period characteristics in Section IV-A.

We evaluate implementation details and required hardware changes in Section V-A. We then compare the expected energy reduction in different implementations and different applications in Section V-B.

A. Implementations and Hardware Overhead

Hardware Overheads. We first estimate the hardware overheads of each encoder using Verilog designs synthesized by the Synopsys toolchain [31] with a 16 nm industrial technology library. Area and delay are estimated through standard-cell synthesis and static timing analysis, and they are presented in a technology-independent manner by normalizing relative to the area and delay of a canonical two-input NAND gate...
Fig. 7: The hardware overheads of the different encoder designs (in NAND2 gate equivalents).

with a drive strength of “1”. Figure 7 shows the area and delay needed for each encoder. All encoding schemes only require eight to ten two-input NAND delays. While the actual delay varies in different process technologies, e.g., DRAM process vs. GPU process, the absolute values of the area and delay are very small. For example, the canonical NAND2 gate we normalize to has an area of 0.156µm² and a delay of 11 ps, such that the largest hardware structure (MTA encoding) has an area of 0.002286mm² and a delay of 0.09 ns in this 16nm technology library. Furthermore, since MTA encoding is currently utilized in GDDR6X, performing an alternate encoding instead of MTA will add little more than a 2:1 mux delay to the timing path. In the unlikely event that the additional delay requires additional cycles of latency to encode and decode each transfer, this performance impact was simulated, and the average degradation was 0.14%. The additional logic area for those encoding schemes is smaller than MTA, which itself is of modest size. If the designer found that the area or delay of these encoders is a concern, DBI could be eliminated to reduce the area even further. The area savings from eliminating DBI increase with the code sparsity, and they range from a 42% savings (4b3s-3) to 86% savings (4b8s-3). The delay of the encoders is cut by more than half without DBI.

There are multiple architectural and implementation choices for sparse encoding schemes, which are tightly coupled with the area and power overhead for their integration.


The first consideration for integrating a SMOREs scheme is whether different sparse encoding schemes can be dynamically selected based on the detected gap between commands. This is called variable code specification. This approach can save more I/O energy by using longer more energy efficient codes when longer gaps exist, but it requires more area in the DRAM and host processor to integrate multiple encoders and decoders. It also requires the additional complexity of tracking longer gaps and correctly enabling the appropriate encoding format.

A simpler alternative approach is to support only one sparse encoding, applicable to the most common situations. This static code specification is much simpler to implement, with only the single additional encoder/decoder and simpler control logic. This mechanism leaves some potential energy savings on the table, however. Based on the profiling results in Section IV-A, we found that more than 73% of all idle periods are one clock gap. Therefore, using a single code selected for this scenario is expected to be the most applicable.

**Exhaustive Gap Detection vs. Conservative Gap Detection.** Another design choice is related to how the idle gap detection scheme handles long gaps between successive commands. Intuitively, logic in the command pipeline can easily detect the interval since the last READ/WRITE request. Dealing with very long gaps is complex, however. For example, in a scenario where there is a huge gap after a READ request, the DRAM may need to start encoding the data to return for first request even before receiving the second request. Since the DRAM has seen a fairly long gap, it would typically encode the result with a sparse encoding. If the next command is a READ, then this would be safe. However, if the next command is a WRITE, using a long sparse code might create a bus conflict with the write data which is typically sent much sooner after the WRITE command than data is returned after a READ command.

The GPU could always assume the read data before a write is encoded and delay the write command and data to avoid the issue. This would effectively increase the read-to-write turn delay, and negatively impact performance. We do not consider this a good solution.

An alternative approach to avoid this situation is to modify the DRAM such that a WRITE command can be sent sufficiently in advance of the write data that the DRAM would have received the WRITE command before the preceding read data was encoded for return to the GPU. Since the DRAM will see a WRITE in time, it can avoid sparsely encoding the preceding read response and avoid the need to delay receiving the write data. Sending the WRITE command further in advance of its data is effectively increasing the “write latency” DRAM timing parameter. Increasing this parameter to send the WRITEs earlier has no performance impact since write data is buffered and deciding to turn the bus around and start issuing writes happens far before a write command is sent. However, there is some area cost as the write command is effectively sent early and must be internally staged in the DRAM. This scheme is called exhaustive gap detection.

A simpler approach is conservative gap detection that tracks the consecutive requests only for several clocks after a received request. We evaluate an eight-clock window. If another request arrives at the DRAM within this predefined detection period, it leverages the idle period for energy reduction. If no other request arrives, it conservatively assumes there is no gap in current requests. This conservative case is required because it must assume a WRITE command will follow. The data will be coded with the default MTA scheme, guaranteeing no bus conflict will occur. There is a small energy saving opportunity lost in these cases, but it is a simple and effective solution.

Combining these two factors of variable/static code specification and exhaustive/conservative gap detection, there are four approaches to SMOREs design integration, which we
evaluate in the next section.

B. Energy Reduction

Figure 8 shows the per-bit energy consumption of data transfers, normalized to two different baselines. In Figure 8a, the baseline is the energy consumption of MTA enabled PAM4 encoding, discarding any postamble energy consumption. As we discussed in Section IV-A, if there is any idle period on the GDDR6X bus, MTA issues a one clock L1 postamble to avoid any potential 3ΔV transition, which increases energy consumption. Therefore, with the current GDDR6X PAM4 signaling, applications with more idle periods show higher energy consumption. Figure 8a plots and sorts the applications compared to the GDDR6X baseline with ascending idle period frequencies. Applications with more idle periods are placed further in the right within each benchmark group. The figure confirms that having more idle periods with MTA consumes more energy, due to the postamble activity. Our SMOREs schemes do not require any postamble and they can leverage any idle periods to reduce energy. Thus, using sparse encodings, applications with more frequent idle periods consume less energy. This plot shows two SMOREs implementations, one of which is detecting all gaps and selecting the best code length, while the other implementation is still detecting all gaps but applying the same 4b3s3 sparse code regardless of the length of idle periods. Both variable and static code selection mechanisms reduce I/O energy significantly by 28.2% and 26.8%, respectively. However, the difference between the two mechanisms' energy reduction is relatively small. Since single-cycle gaps predominate, using the encoding scheme optimized for this case enables most of benefit that can be achieved with more complicated mechanisms.

Figure 8b shows the relative per-bit energy consumption of our mechanisms compared to an optimized MTA enabled PAM4 signal without postamble energy. A hypothetical modification to MTA encoding could apply a level-shifting approach similar to the sparse encoding scheme to avoid 3ΔV transitions when transitioning to idle with much less energy than the baseline postamble scheme.

We also evaluate a conservative gap detection implementation, which we compare against the exhaustive gap detection case in Table V. We observe that the energy saving drop with the simpler, conservative approach is small (1.6%). This is again due to the dominant one clock idle period in all of the applications.

The DRAM data transfer energy is only a portion of the total DRAM energy. Across the workloads we study the
TABLE V: Energy saving with different SMOREs schemes.

<table>
<thead>
<tr>
<th>gap detection</th>
<th>code specification</th>
<th>energy saving</th>
</tr>
</thead>
<tbody>
<tr>
<td>exhaustive</td>
<td>variable (4b/3:8jS-3)</td>
<td>28.2%</td>
</tr>
<tr>
<td>exhaustive</td>
<td>static (4b3S-3)</td>
<td>26.8%</td>
</tr>
<tr>
<td>conservative (8 clocks)</td>
<td>static (4b3S-3)</td>
<td>25.2%</td>
</tr>
</tbody>
</table>

baseline GDDR6X data transfer energy is 706.9 fJ/bit plus an additional 10 fJ/bit for the MTA encoder/decoder logic. Micron reports the average energy required for the entire DRAM is 7.25 pJ/bit [13]. Thus, the data transfer energy represents about 10% of the GDDR6X DRAM power. The power savings introduced with our proposed encoding scheme including the 3.5 fJ/bit for the 4b3s-DBI encoder logic, represents 2.5% of the total DRAM power. Future DRAM architectures that focus on reduced core power, like LPDDR family devices or other proposed low-power DRAM architectures [23], would see more significant savings as a result of these data transfer energy reductions.

VI. CONCLUSION

PAM4 encoding allows scaling the bandwidth of high-performance DRAM interfaces without necessitating higher signaling frequencies. However, existing conventional bus encoding schemes that look to enhance energy-efficiency are not always directly applicable, nor fully effective on PAM4 interfaces. In this work, we showed that sparse encodings can be used to increase the energy-efficiency of the GDDR6X PAM4 interface by utilizing sparse codes, which are applied opportunistically to lengthen the data transfers between the DRAM and GPU to fill available idle cycles between memory requests. We showed that despite the generally high DRAM utilization by GPUs, several such opportunities do indeed exist, and a carefully choosing the particular encoding scheme can yield significant I/O interface savings without any performance overheads. This family of dynamic energy-efficient coding schemes, while applied to the GPU-DRAM interface in this work, can also form the basis of energy-efficient signaling between different chips/chiplets in emerging multi-chip-module (MCM) chips.

REFERENCES


